A Numerical Engine for Distributed Sparse Matrices

by

Ricardo Telichevesky

B.Sc., Universidade Federal do Rio Grande do Sul (1985)
M.Sc., The Technion - Israel Institute of Technology (1988)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1994

Signature of Author____
Department of Electrical Engineering and Computer Science

Certified by____
Jacob K. White, Associate Professor of Electrical Engineering
Thesis Advisor

Certified by____
William J. Dally, Associate Professor of Computer Science and Engineering
Thesis Advisor

Accepted by____
Frederic R. Morgenthaler, Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

A Numerical Engine for Distributed Sparse Matrices

by

Ricardo Telichevesky

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 2000, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In fields as diverse as electronic circuit design, fluid dynamics, and structural analysis, the behavior of complex systems is modeled by large, sparsely coupled systems of differential equations. Numerical solution of such systems is computationally expensive because even the most efficient algorithms evaluate and factor large sparse matrices hundreds or thousands of times, and on most general purpose computers sparse matrix operations are inefficient. Part of the problem is that accessing sparse matrix elements is complicated, resulting in a poor utilization of computational resources. The irregular non-zero pattern makes it very difficult to parallelize the operations; and the non-uniform data access time makes pipelining very inefficient.

This thesis suggests exploiting the infrequent change in matrix structure by developing a symbolic compiler, and a special purpose parallel computer that uses compiler clues to accelerate sparse data access. The compiler combines partitioning, scheduling, and storage allocation algorithms in order to exploit locality of reference, achieve a high degree of parallelism, and simplify the operand access in the sparse matrix, which in turn insures efficient pipelining. Each processing element contains a specialized datapath consisting of multiple interleaved memories and functional units, and a microprogrammed control unit capable of initiating several datapath operations per clock cycle. Extensive behavioral and register transfer level (RTL) emulation of the execution of `SIMLAB`, a circuit simulation program, suggests that the combination of these hardware and software techniques yield a high degree of utilization of computational resources both in the assembly of circuit equations using device models and in its associated sparse matrix solution.

Thesis Supervisors: Jacob K. White, Associate Professor of Electrical Engineering
William J. Dally, Associate Professor of Computer Science and Engineering

Acknowledgments

This thesis is based on work supported in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq-Brazil) and the Office of Naval Research.

This thesis results from a combined effort with Professors Jacob White and William Dally. More than providing advice, support, and guidance, their ideas are an essential part of this research.

I wish to thank all the members of the research group that have helped me in this research. Miguel Silveira, Keith Nabors, Kevin Lam, Khalid Rahmat, and Don Baltus are notable collaborators in the actual implementation of the algorithms presented in this thesis, as well as in the architectural design. Steven McCormick provided the essential macros that helped so much in writing the thesis. Bob Armstrong, our UNIX guru, was always there to answer all sorts of system-related questions.

Prathima Agrawal and John Trotter of AT&T Bell Labs have been involved in the early stages of this research and contributed significantly to the development of scheduling and partitioning heuristics. Greg Papadopoulos also took time to participate on my thesis committee. Milena Levak helped so much in all the complicated requirements of the International Students Office. Many thanks to Marilyn Pierce, a saint that by saying "No problem" miraculously keeps the sanity of graduate students.

Thanks to José Monteiro, Mattan Kamon, Ignacio McQuirk, Filip Van Aelten, Songmin Kim, Jennifer Lloyd, Xuejun Cai, Stan Liao, Amelia Shen, Chris Umminger, Joel Phillips, and all people in the eighth floor for their invaluable assistance.

Thanks to my friends Rodrigo Paiva, Julia Allen, Cristina Lopes, Brian Pan, Chris Berry, and Paul Yu for the encouragement, patience and help in so many aspects of my graduate student life here in the United States. From Brazil, the encouragement and emotional support of my long-time friends Ricardo Mester, Jairo Moscovich, Nelson Zamel and of my family made it all possible.

Contents

List of Figures

List of Tables

1

Introduction

Many physical systems can be modeled by large sparse systems of differential equations. Ordinary differential equations (ODE's) are often used to model time varying discrete systems such as electric circuits, or to predict the motion of planets or other bodies in space. Partial differential equations (PDE's) are used to model the space and time variation of continuous systems such as fluid and heat flow and electron transport in semiconductors. Most of the time for the numerical solution of differential equations is spent evaluating non-linearities and solving a sparse set of linear equations.

1.1 Thesis Contributions

In order to efficiently evaluate the non-linearities and solve the sparse set of linear equations associated with the numerical solution of differential equations, this thesis describes the development of the Numerical Engine, a special purpose parallel processor that would take advantage of specialized hardware and software precompilation techniques in order to accelerate the solution of sparse sets of linear equations. Even though the techniques discussed in this thesis can be extended to the numerical solution of differential equations in general, it is focused on the simulation of electrical circuits, a specially challenging problem due to its unstructured nature.

In order to demonstrate the practical utilization of the Numerical Engine, we developed a modified version of a circuit simulation program, `SIMLAB`. Most of the computation time spent during the execution of `SIMLAB` is split between the assembling of sparse circuit equations, and their solution using sparse matrix techniques.

One important characteristic of these computations is that several iterations are executed with different numerical values but the sparse structure of the equations and the connectivity of the circuit is kept static. We can take advantage of this fact by executing a *symbolic compilation* that generates addressing clues, partitions the matrix and device data, schedules the internal tasks for each individual processor, and schedules the interprocessor communication sequence for the target multiprocessor system. Figure 1-1(a) depicts the traditional SIMLAB flow while the symbolic compilation step for multiprocessor execution on the Numerical Engine is introduced in Figure 1-1(b).

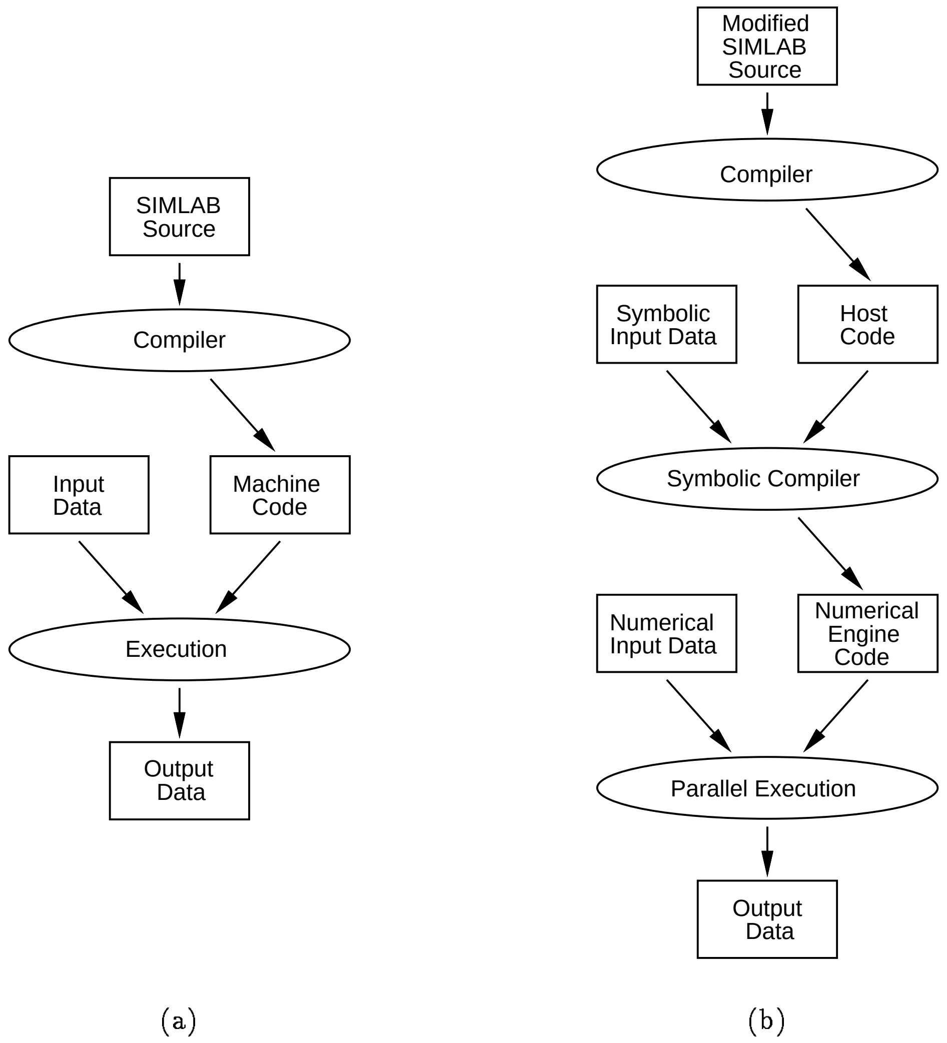


FIGURE 1-1: Introducing a symbolic precompilation step for SIMLAB

The major contributions of this thesis in order to speed up the parallel execution of the matrix assembly and sparse matrix decomposition tasks are:

- The O^2SA technique, which combines scheduling and storage allocation algorithms to enhance the processor efficiency. The essence of this technique is to keep an active data subset, necessary to achieve a high degree of concurrency in the sparse matrix factorization, in a small, fast memory tightly coupled with the floating point unit. Section 2.5 further discusses how to use this technique to substantially increase the factorization speed.

- Exploit the properties of sparse matrices and scattered arrays to efficiently use memory interleaving as a general technique that increases the processor-memory communication bandwidth. These properties are further discussed in Section 2.2.1.
- $O(V \log V)$ scheduling heuristics to efficiently solve sparse matrix factorization on several pipelined processors connected through a high speed bus. The fast scheduler is able to exploit the underlying hardware characteristics such as interleaving, pipelining, static column misses in DRAMs, bus conflicts, etc. A detailed discussion of the scheduling schemes proposed is presented in Section 2.3.
- Software techniques for the parallel device evaluation and parallel stamping of device contributions, in order to assemble the sparse set of equations representing the circuit behavior. Even though these techniques are fairly general and can be used in any distributed memory multiprocessor system, they have been specially tailored for the Numerical Engine. A detailed description of these techniques is available in Chapter 3.
- Specialized datapath and control in each Numerical Engine processor that permits the execution of several tasks per clock cycle. These tasks include multiple memory access, concurrent floating point and address generation operations, conditional branching, and loop control. The processor architecture is discussed in detail in Chapter 4.

In order to introduce the reader to the thesis material, the next section provides an overview of the numerical solution of differential equations.

1.2 Numerical Solution of Differential Equations

Numerical techniques for solving differential equations often discretize time or space (or both) to reduce the differential problem to a sequence of large, usually non-linear, algebraic systems. These systems, in turn, are solved by an iterative technique, such as the Newton-Raphson method [Press92] that involves linearizing the non-linearities, and then solving the linearized system of equations.

The evaluation of the non-linearities in a system of n equations in n unknowns, in the form of a dense matrix A, takes $O(n^2)$ operations; the solution of the linearized system takes $O(n^3)$ operations. These operations would be prohibitively expensive for systems with more than a few hundred discrete elements. In many cases, however, the sparse connectivity of the discrete elements causes most of the entries in the matrix A to be zero, and only a small percentage (usually less than 1%) need to be stored and operated on. The matrix A is said to be sparse, and by exploiting its sparsity the solution can be found in a tiny fraction of n^3 operations. Also, the cost of assembling the matrix A is reduced in many cases to $O(nonzeros)$.

The simulation of electric circuits is a good example of sparse matrix usage. In the modified version of `SIMLAB`[Lumsdaine90] (a circuit simulation program similar to `SPICE`[Nagel75]) used in this thesis, each node voltage not connected to an independent voltage source is represented by one unknown and the matrix has a non-zero entry a_{ij} only if there is a circuit element connecting node i and node j. For example, the matrix *iir12*, depicted in Figure 1-2, was created during the transient simulation of the first and the second stages of an infinite impulse response digital filter. *iir12* has $7,310$ equations and after the decomposition it has only $153,858$ non-zero elements, corresponding to a nonzero density of 0.28%.

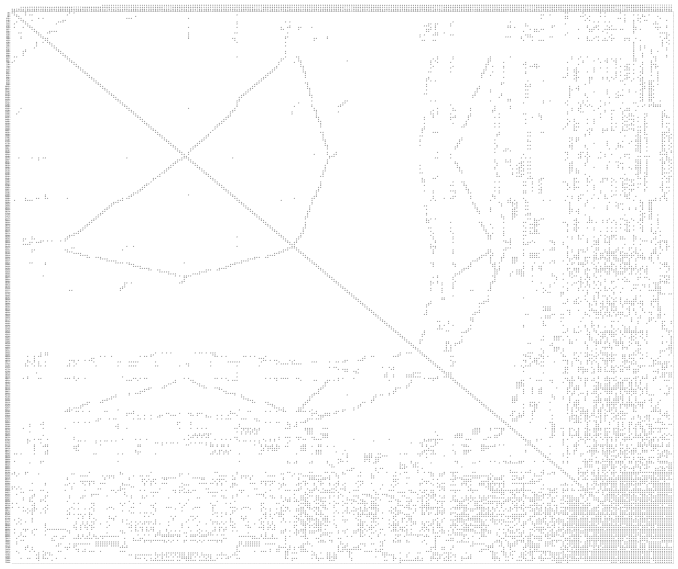


FIGURE 1-2: Sparse matrix created during the simulation of the first and second stages of an infinite impulse response digital filter

For PDE's, the spatial discretization of the problem in a grid naturally leads to sparse matrices. Finite difference methods will often generate regular sparse matrices given they are often linked with structured grids, while finite element methods will tend to generate an irregular non-zero pattern due to the lack of grid structure.

An important issue in the numerical solution of differential equations is the method used for solving the linear set of equations. *Direct methods*, like *LU (lower-upper) decomposition*, are able to solve the system in a fixed and finite number of steps, but are often very computationally expensive and require large amounts of memory. *Indirect* or *iterative methods*, like the *Gauss-Seidel* relaxation [Golub89], produce an infinite sequence of approximate answers that may or may not converge to the correct answer. If the proper conditions on the matrix A are satisfied, which is usually the case in many computational fluid dynamics problems, iterative methods will often compute a solution within a defined error margin in a fraction of the time spent by direct methods and typically will not require as much memory. However, we are particularly interested in the solution of sparse matrices related to circuit simulation, which many times exhibit poor characteristics for an iterative solution. Most commercially available circuit simulation programs available today rely on direct methods for sparse matrix solution.

Given these constraints, this thesis focuses on direct methods, and more specifically, focuses on LU decomposition, which in almost all cases is able to compute the correct solution, if one exists.

This thesis focuses on hardware and software techniques for the efficient factorization of sparse matrices and the parallel evaluation of non-linearities in an electrical circuit. The next two sections provide a brief introduction and present some background material on these topics.

1.3 Parallel Sparse Matrix Decomposition

The major difficulty associated with LU decomposition is its superlinear time complexity — the solver tends to dominate the simulation time for large problems. This difficulty, along with the fact that most circuit simulators contain in their core a sparse matrix package that performs LU decomposition, explains the great interest in exploiting the speed of parallel and pipelined machines in order to accelerate the algorithm execution. This task is difficult due to the irregular structure of most sparse matrices associated with electrical circuits, which demands a very complex sequence of instructions and data access. Many techniques are reported in the literature to overcome these difficulties. In the following sections, we will briefly discuss *reordering*, *storage organization* and *scheduling.*

1.3.1 Reordering

One aspect that makes the utilization of LU decomposition on a sparse matrix difficult is that even though a given element a_{ij} is zero initially, some operations can make a_{ij} non-zero during the decomposition. The new non-zero element is usually referred to as a *fill-in.* The reordering of the rows and columns of a matrix can strongly affect the number of fill-ins, and consequently the computational cost of the LU decomposition [Duff86].

Reordering is an operation that could be performed on the symbolic structure to obtain a minimum number of fill-ins. This problem has been shown to be "NP complete" [Rose78, Yannakakis81]. Many heuristics have been proposed, but the Markowitz [Markowitz57] reordering method is generally preferred since it produces on average no more than 5% more fill-ins that the best of the other methods, but takes much less time. It is a greedy strategy that tries to minimize the number of fill-ins that can occur at each step. Empirical results show that Markowitz reordering usually takes between one and two orders of magnitude more time than the actual numerical decomposition. For this reason, most sparse matrix implementations that do not require numerical pivoting perform Markowitz reordering on symbolic data only once in the beginning of the program execution.

In a parallel processor, the objective of the ordering algorithm should be the reduction of the completion time for the decomposition, which is not necessarily the ordering with the smallest number of operations. A good figure of merit for a reordering algorithm in a multiprocessor system is the minimum number of steps required to compute the solution given an infinite number of computational resources. Several ordering algorithms have been proposed which attempt to minimize the number of steps without increasing substantially the total number of operations [Huang79, Smart89, Chang88].

1.3.2 Storage Organization

Early versions of sparse matrix codes used linked list structures to represent the non-zero elements in the matrix. Given a matrix element a_{kj} (let us call it *source*), in order to perform an operation like $a_{ij} = a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj}$, it was necessary to traverse the linked list of row i, find the matching element a_{ij}, and then perform the arithmetic operation. Since these type operations dominate the LU factorization time, the slow matching process makes the entire code very slow. Even worse, the irregular access time of the elements made pipelining very inefficient.

The *Scatter-Gather* approach starts by scattering the elements of a destination row i into a vector of size n. Computers with pipelined indirect addressing could then perform the source-destination match in constant time. After all the updates to row i are finished, the elements are gathered back in a dense vector. This technique has been used in the YSMP code [Eisenstat77]. One of the disadvantages of this approach is the Scatter-Gather overhead, which is $O(nonzeros)$. However, the major problem with this technique is that its extension to a multiprocessor environment limits the available parallelism too much, as we shall demonstrate in Section 2.4.

Stager [Stager87] proposed the enumeration of all elemental operations and addresses required for the factorization. Even thought this approach can fully exploit pipelining, the excessive memory requirement makes it infeasible for large matrices, as the number of operations grow superlinearly with the size of the matrix.

Sadayappan [Sadayappan89] studied the implementation of the sparse matrix factorization by *Overlap-Scattering.* All destination rows are scattered in memory, allowing a source-destination match of any element in constant time, just like in the Scatter-Gather approach, but no scatter and gather operations are required. In practice, if independent scatter-vectors of size n are used, the total space required would be prohibitive ($O(n^2)$). Fortunately, the sparsity of the rows can be exploited to share the memory efficiently, by overlapping them in such a way that the non-zeros of one row would *fill-up* the nonused elements of another. The rows are placed in scattered form into a single linear array, with different origins, such that no two rows have a non-zero in the same location in the array. During the update operation, the matching target address is computed with simple addition. Then, data from the target row is read, processed and written back to the same memory location.

Perhaps the most significant drawback in the usage of overlapped-scattered arrays is that each elemental update operation requires two accesses to a large memory that stores the entire matrix. This thesis proposes a novel technique that combines sophisticated scheduling and storage allocation algorithms by keeping only a selected set of *active* overlapped-scattered rows in a small, fast access memory in order to further enhance the processor speed. We shall refer to this allocation structure as an *Overlapped-overlapped Scattered Array* (O^2SA). This technique is specially relevant when applied to modern computers, as the overall performance of these systems is strongly linked with the proper cache utilization.

In Section 2.4 we intend to discuss the overhead involved in using the overlap-scatter structure, and how special purpose hardware can be built in order to speed up the data access if the matrix is stored in the overlap-scattered form. Section 2.5 discusses in detail the O^2SA structure, and also discusses a processor architecture designed to take advantage of O^2SA in order to further accelerate the matrix factorization.

1.3.3 Scheduling

In order to improve the utilization of a multiprocessor (or pipelined) system, a good scheduling algorithm is necessary to *choose* among the possible operations so that the most critical is executed first. The scheduling problem has been shown to be NP-complete and several heuristics have been proposed to make it computationally feasible for large problems.

Wing and Huang [Wing80] modeled the sparse matrix factorization by an acyclic directed graph in which nodes represent a task, in this case an elemental arithmetic operations applied to the elements of A, and the arcs represent the precedence relations that exist among the operations in the factorization process. They applied Hu's [Hu61] level scheduling strategy and found that the speed-up using parallel processing is proportional to the number of processors when it is ten to twenty percent of n. The major problem with their model is that the scheduling is associated with the enumeration of all elemental operations, a requirement that makes it infeasible for large matrices. As a side aspect, their model does not take into account interprocessor communication, and the different times consumed by different arithmetic operations.

Sadayappan and Visvanathan [Sadayappan88] treat the problem at differents levels of granularity. If individual tasks are row operations, then the task graph is referred to as a medium-grained model, while choosing the tasks to be elementary arithmetic operations results in a so called fine-grained model. They implemented a parallel sparse matrix solver in a shared memory multiprocessor and report that the medium-grained approach is consistently superior for large matrices due to lower operand access costs and better vectorization potential. The greedy heuristic used in Sadayappan's work tries to schedule tasks as soon as they become enabled. Even though this simple scheduling can be executed very fast, its results are quite poor since the greedy scheduling mechanism lacks the information about the *priority* of the execution of certain critical tasks.

Trotter and Agrawal [Trotter90a] suggested a level-based approach, in which all tasks of a certain level of a task graph would be executed, and then the computation would proceed in the next level. The scheduling results are not as good as the greedy-based approach because at each level many processors may stay idle. This happens because one processor is not permitted to start the computations at the next level until all the other processors had finished the computations at the present level.

Several algorithms based on critical path scheduling are reported in the literature. In this scheme, the first scheduled tasks are in the critical path. Next, each branch is analyzed and the local critical path is scheduled in a recursive fashion. Even though this scheme produces results that are consistently better than level scheduling, the time complexity of this algorithm is $O(E(V+E))$ where E represents the number of edges and V the number of vertices in the task graph. For large matrices, the schedule time using this scheme becomes prohibitive.

We presented an $O(VlogV)$ scheduling heuristic based upon the measure of the *remaining completion time* [Telichevesky91b]. The resulting schedules are consistently better than the level-based and greedy heuristics without a substantial penalty in the execution time. We will further discuss this scheduling heuristic and present simulation results in Section 2.3.

1.4 Parallel Circuit Simulation

In order to accelerate the circuit simulation, the parallel assembly of the network equations representing the behavior of the circuit is as important as the fast parallel sparse matrix decomposition. The parallel assembly of the equations consists of evaluating the device models and adding or *stamping* their contributions to formulate the equations.

For example, let us consider a circuit simulator like SIMLAB. Each timestep in a transient analysis involves a Newton-Raphson iteration in which a voltage vector v^k can be computed iteratively by evaluating the non-linearities and solving a sparse set of linear equations of the form:

$$J_F(v^k)(v^k - v^{k-1}) = -f(v^k) \tag{1.1}$$

where $J_F(v^k)$ is referred to as the *Jacobian matrix* that consists of the partial derivatives of the components of the charge/current equations with respect to the components of v^k, and $-f(v^k)$, the *right-hand-side vector*, consists of the contributions of the current and charge balance to the error in the approximation of the iterate v^k. The determination of the terminal currents and charges and their derivatives in respect with the terminal voltages for every device instance in the circuit is usually referred to as *model* or *device evaluation*. The sum of these contributions into the Jacobian matrix entries and into the right-hand-side vector in order to assemble the network equations is usually called *stamping*.

In order to achieve high efficiency for the evaluation and stamping tasks, it is necessary to consider them both at processor level and in the context of multiprocessing.

A large circuit often has thousands of circuit element instances. The model evaluation can be executed locally in each processor, so that we could simultaneously evaluate several elements. In order to quickly execute these tasks each processor of the Numerical Engine contains several functional units controlled by a single, wide instruction stream able to initiate several operations per clock cycle. Hand programming such a machine is not a trivial task, and building a simplified compiler that makes this job easier might be even more difficult. Even though some background work in this field is presented in [Ellis85], and we performed some preliminary studies in order to implement such compiler, the task was abandoned. Consequently, the parallel model evaluation was only simulated at the behavioral level, discussed in detail in Section 4.3.1.

Stamping requires a more complex analysis in the context of multiprocessing. Sadayappan and Visvanathan [Sadayappan88] describe this process as a *lock-synchronized parallel loop.* They present a theoretical analysis and compare with measured speedups on a shared memory multiprocessor. However, the implementation becomes inefficient if the number of parallel processors is larger than six. We intend to further discuss this problem and present a detailed account of our implementation on the Numerical Engine (a distributed memory system) in Chapter 3.

The computation of transcendental functions is a very time consuming task in non-linear evaluation. A mathematical library of transcendental functions based on the fast evaluation of Chebyshev polynomials [Clenshaw63, Clenshaw62] was hand-coded for the Numerical Engine architecture proposed in Chapter 4. A combination of operation reordering, clever register allocation and multiple memory access was used to achieve a high utilization of the floating point hardware. We shall further discuss these techniques in Appendix A.

1.5 Numerical Engine Architecture

The factorization of sparse matrices is a very inefficient operation on general purpose computers due to the difficult access to sparse matrix elements. The difficulty in accessing sparse matrix elements is partly due to the address generation complexity, and partly due to the large bandwidth requirements between the main memory and the floating point unit. The end result is a poor utilization of the floating point resources, specially in state-of-the-art heavily pipelined architectures. For instance, experimental results show that the average floating point unit utilization of an AXP ALPHA computer is around 2% for sparse matrix factorization. A detailed discussion of the experimental results is provided in Section 4.4.

The Numerical Engine architecture overcomes the difficulties mentioned above by using specialized hardware and the combination of scheduling and storage algorithms for the fast parallel factorization of sparse matrices. The primary objective of the architecture development was the design of a multicomputer containing processing elements with added hardware support for fast sparse execution of operations of the type $a_{ij} = a_{ij} - a_{ik} \times a_{kj}$, which are the most frequent operations in matrix decomposition. In order to achieve its primary objective, each processing element has multiple interleaved memories to supply data at high rates for the floating point unit, support for the concurrent generation of addresses and writeback, and a simple but fast pipeline interlock mechanism. In order to provide high speed control with the smallest possible latency, the controller is a very simple microprogrammed unit, containing a very wide microinstruction memory tightly coupled with the datapath. The Numerical Engine performance for sparse matrix decomposition has been tested by a detailed register transfer level (RTL) simulation, using a hand microcoded sparse matrix package for a single processor. The results indicate that a single Numerical Engine processor could solve sparse matrices at a sustained rate of 73 MFlops, or 73% utilization of the floating point unit. These results are further discussed in Section 4.4.

The Numerical Engine is also expected to perform the device evaluation and stamping tasks described briefly in Section 1.4 and discussed in detail in Chapter 3 with a high degree of floating point unit utilization. Even though these tasks have not been simulated in detail using RTL simulation due to the extreme difficulty in writing its microcode, it seems that they could also achieve high performance. This theory is supported by the high degree of floating point unit utilization, 89% or 89 MFlops, achieved in the RTL simulation of the evaluation of the transcendental function e^x, described in detail in the Appendix A.

A detailed description of the Numerical Engine architecture is presented in Chapter 4.

1.5.1 Related Work

Gyurcsik and Pederson [Gyurcsik85] have designed, assembled and tested a prototype of an attached processor for an IBM-PC computer for MOS model evaluation. The system uses a table-based approach for the fast evaluation of the contributions of MOS transistors to the assembly of the network equations.

Ginosar and Jacobson [Ginosar85] proposed a VLSI architecture for circuit simulation based on waveform-relaxation. They proposed several specialized building blocks that would perform concurrently different tasks required for the implementation of the waveform-relaxation algorithm.

Lewis [Lewis86] suggested the usage of a Forward-Euler integration method for avoiding the solution of a sparse system of network equations in circuit simulation and proposed a specialized hardware for matrix multiplication and table-based evaluation of the device contributions. The shortcoming of the Forward-Euler integration method is that it requires very small timesteps to ensure the stability of the solution.

Nakata, Tanabe, Onozuka, Kurobe and Koike [Nakata87] used a relaxation method for circuit simulation, dividing the circuit to be simulated into several *modules* for independent evaluation on a specialized parallel computer.

Agrawal and Trotter [Agrawal92] have designed, assembled and tested a prototype machine, named PACE (Parallel Architecture for Circuit Evaluation), which consists of four *Intel i860* [Intel90] microprocessors connected by a wide high speed bus. The PACE hardware is intended for circuit simulation using a direct method solver. The PACE partitioning and scheduling heuristics for sparse matrix decomposition are the same as the techniques presented in Chapter 2 of this thesis.

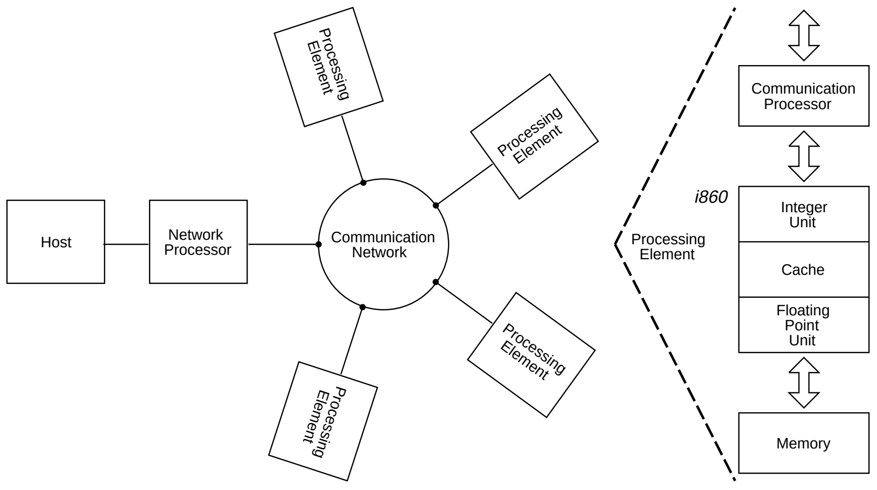


FIGURE 1-3: General architecture of the PACE

Figure 1-3 gives a block diagram of the machine. It consists of several processing elements (PEs) and a communication network (CN) controlled by a network processor (NP) and a host processor. The prototype PACE is a wire-wrapped circuit board that connects to the backplane of a VME SUN workstation. The network processor provides an interface to the host and controls the PEs and the communication network by sequencing the communication transactions. Each of the four PEs has a communication processor that interfaces to the communication network, an integer unit that is responsible for address calculation, a floating point unit, cache, and a memory system to hold the bulk of matrix data.

Performance measurements on PACE, using matrices generated during the simulation of VLSI circuits, demonstrate the effectiveness of the partitioning and scheduling heuristics suggested in this thesis. A detailed discussion of the PACE performance is provided in Section 2.6.2.

1.6 Thesis Outline

As discussed before, this thesis presents hardware and software techniques for the fast numerical solution of differential equations, with an emphasis on the fast execution of the circuit simulation algorithm. Most of the computation time in a circuit simulator is spent in the assembly of a sparse set of linearized equations that represent the circuit behavior and solving them.

Chapter 2 describes several software and hardware techniques that accelerate the solution of sparse sets of linear equations. These techniques include storage allocation methods to simplify data access, multiprocessor scheduling mechanisms, and how to exploit the locality of reference and hardware characteristics such as pipelining, main memory paged-mode access, and interleaving, in order to achieve a high degree of utilization of computational resources both at processor and at system level.

Chapter 3 describes several software techniques that accelerate the assembly of the linearized equations representing the circuit behavior in a general purpose multiprocessor environment, and are specially effective when applied to the Numerical Engine proposed in Chapter 4.

Chapter 4 provides a comprehensive description of the proposed Numerical Engine architecture, both at system and at processor level. A complete description of the interprocessor bus interface protocol and electrical specification, as well as system-wide issues such as clock generation is provided in the first section of Chapter 4. The second section presents a detailed explanation of the register-transfer-level operation and accurate timing information on the components of each individual processing element. The third section provides a description of software developed for the architectural emulation and the verification of its proper operation when executing a modified version of `SIMLAB`. The last section provides a comparison of the predicted performance of a single processing element with measurements taken from commercially available state-of-the-art general purpose computers.

Finally, Chapter 5 summarizes the simulation results and presents a discussion on the major contributions of this thesis, suggesting some paths for future research.

2

Parallel Sparse Matrix Decomposition

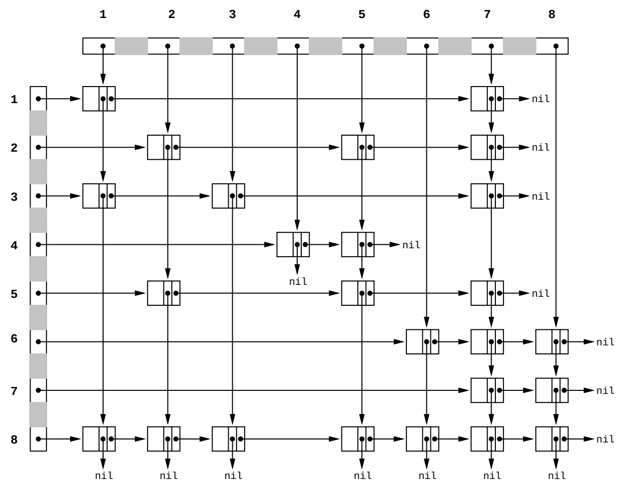
A sparse system of linear algebraic equations can be written in matrix form as $Ax = b$, where $A = [a_{ij}]$ is an $n \times n$ sparse matrix of real or complex coefficients, x is the vector of n unknowns and b is the vector of n known right-hand side terms. In this work, we will assume a_{ij} are real numbers.

If a_{ij} are complex coefficients, the sparse matrix decomposition can be performed using analogous techniques. The overall pattern of matrix data access and operations, and specially its impact on computer architectures is very similar. In fact, matrices with complex elements represent a less challenging problem in terms of achieving high processor efficiency. One of the major factors that reduce the efficiency of sparse matrix computations is the bandwidth between the memory and the floating point unit. It is very difficult to design a memory system that can keep pace with the floating point unit for the predominant operation in matrix decomposition, called *gaxpy* [Golub89], and defined as $a_{ij} = a_{ij} + a_{ik} \times a_{kj}$. In sparse matrices with real coefficients, four memory accesses are required per *gaxpy*, and only two arithmetic operations are performed. If the coefficient are complex, eight memory accesses are required, but eight arithmetic operations are performed.

It is desirable to exploit the sparsity of some matrices, such as the large, unstructured matrices generated during circuit simulation. The most obvious reason is to avoid storing the elements that are zero in the beginning of the decomposition and will never change during the decomposition, which are called *structural zeros*. Not storing these zeros makes it much more complicated to store the other matrix elements, called *structural non-zeros*. These elements could be originally non-zeros, or they might become non-zero during the course of decomposition, and in that case they are called *fill-ins*. In the matrix examples further discussed in this chapter, the non-zeros account roughly for 0.1% to 0.7% of the elements in the corresponding dense matrix. Another advantage of not storing the structural zeros is that no time is required to access them, which in turn avoids the trivial multiply-by-zero and add-with-zero operations.

$$\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & X & & & & & & X & \\
2 & & X & & & X & & X & \\
3 & X & & X & & & & X & \\
4 & & & & X & X & & & \\
5 & & X & & & X & & X & \\
6 & & & & & & X & X & X \\
7 & & & & & & & X & X \\
8 & X & X & X & & X & X & X & X
\end{array}$$

(a)



(b)

FIGURE 2-1: A sparse matrix (a) and its associated orthogonal linked list storage (b)

One way of storing the structural non-zeros is by using *orthogonal linked lists*, described in [Kundert86]. For example, consider the sparse matrix depicted in Figure 2-1(a) and its associated orthogonal linked list storage representation, as shown in Figure 2-1(b). In this representation, each structural non-zero element corresponds to a data structure which typically contains the numerical value a_{ij}, the row and column indices i and j, and a pair of pointers to the next data structure in the row and column. Even though simple and very flexible, this orthogonal linked lists are not very efficient when used to perform factorization. Several different storage data structures have been proposed that help to achieve faster access to matrix data, and these will be discussed later in this chapter.

As mentioned in Chapter 1, there are two classes of methods for solving systems of equations: *direct methods*, which are able to solve the system in a fixed and finite number of steps, and *indirect or iterative methods*, which produce an infinite sequence of approximate answers that may converge to the correct answer. Since we are particularly interested in the decomposition of sparse matrices for circuit simulation, iterative methods will only converge if rather strong conditions on A are satisfied.

It is often desirable to solve a system of equations with more than one right-hand side, preserving the original matrix. In circuit simulation, this technique is widely used to avoid the costly reconstruction of the Jacobian matrix in the inner loop of a Newton iteration, as described in Chapter 3. A simple modification of the original Gaussian Elimination scheme, allows the solution of various problems with varying right-hand side vectors b. This technique, called LU decomposition, is widely used in simulators and will be further discussed in Section

2.1. This work is focused on direct methods, and in particular, LU decomposition.

In this chapter, several aspects of parallel sparse matrix decomposition will be discussed. Section 2.1 introduces the basic sparse LU factorization and the column match problem related to the usage of the orthogonal linked list storage by itself. Section 2.2 will describe the Scatter-Gather approach, which solves the column match problem. Section 2.3 will introduce the readers to the extensions of the basic LU decomposition algorithm for a multicomputer environment, and will present a greedy heuristic based on a multitask graph measure called remaining completion time which empirically generates excellent multiprocessor schedulings without a substantial penalty in the execution time. Section 2.4 will describe Overlap-Scatter Arrays, and how its flexibility greatly enhances the utilization of a parallel processor when compared with the plain Scatter-Gather approach. Section 2.5 will introduce O^2SA arrays and how they can be used to exploit locality of reference within each processor without sacrificing multiprocessor performance. Section 2.5.3 also discusses how to change the scheduling mechanism to accommodate different storage organizations. Finally, Section 2.6 addresses some actual implementation issues, using an enhanced scheduler to predict the performance of serial and parallel processors.

2.1 Sparse LU Factorization

The LU factorization or decomposition consists of applying preserving transformations on the original matrix A, decomposing into the product of two matrices, L and U, which are respectively lower and upper triangular, as:

$$Ax = L(Ux) = Ly = b \tag{2.1}$$

Once the matrix A is factored into L and U, it is easy to solve the lower triangular problem $Ly = b$ for y, in a process called *forward elimination*. Then, the upper triangular system $Ux = y$ can be solved for x, in a process called *back substitution*. The decomposition can be thought as the product:

$$\begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

If the matrix is dense, this system has n^2 equations corresponding to the elements of A, and $n^2 + n$ unknowns, the entries of L and U. Since this is an under-determined system of equations, n elements can be chosen freely. For implementation reasons, we chose to set the diagonal elements of U to one, which is often referred to as *Crout's algorithm for LU decomposition*, with the small difference that only row-wise operations are performed.

Assuming the original coefficients a_{ij} are no longer necessary, it is possible to perform all computations in place, i.e. no additional storage is required to store the LU coefficients. The upper-triangular diagonal elements u_{kk} are implicitly stored, as they are all set to $u_{kk} = 1$. In terms of memory usage, these are the key advantages of the LU decomposition method.

Since floating point divisions are more expensive operations than floating point adds or multiplies, another small change in the original Crout's algorithm involves computing the reciprocal of $a_{kk}^{-1} = \frac{1}{a_{kk}}$ and storing it for future use. The original value a_{kk} is discarded.

Given A, a non-singular sparse matrix, Algorithm 2.1 can be used to solve the system of equations $Ax = b$. This algorithm is called *Source-row directed* because at each LU decomposition step, some row k is picked to be used as a *source*, i.e. to update the rows i underneath it. The rows i, updated by row k, are called *destination*, or *target* rows. If it is necessary to solve for multiple $b^{(m)}$, multiple forward eliminations and back substitutions are executed, but only one LU decomposition. The double loop marked "Update" consumes most of the time during the algorithm execution.

Algorithm 2.1 exploits the sparsity of the matrix to reduce the number of operations required to solve the system of equations. If a sparse matrix problem is solved as a dense matrix problem, $O(n^3)$ operations would be required. The number of operations necessary when the matrix sparsity is exploited decreases to $O(n^\alpha)$, where α is a factor which depends on the structure of the matrix. α is 1.5 for a grid, and probably smaller for circuit problems, but difficult to determine exactly. The matrix sparsity is represented in Algorithm 2.1 by the operator **foreach**, which visits all the elements on a list of structural non-zeros. Using the data structure shown in Figure 2-1(b), the access of the desired non-zero elements consists of visiting the next element in the linked list, and is reasonably trivial for all **foreach** cases, except for the element marked a_{ij}^* in the update loop.

Consider, for example, the execution of the update loop in Algorithm 2.1, and more specifically, the operation $a_{87} = a_{87} - a_{81} \times a_{17}$ during the factorization of the example matrix depicted in Figure 2-1. The element $a_{ik} = a_{81}$ is naturally accessed by visiting the next element in the first column list during the execution of the outer **foreach** loop. The element $a_{kj} = a_{17}$ is also easily accessed by visiting the next element in the first row during the execution of the inner **foreach** loop. However, there is no obvious way of accessing directly $a_{ij}^* = a_{87}$ with this data structure. In order to reach element a_{87} it is necessary to traverse the linked list binding the elements of row 8 (or column 7), and checking the column index (or row index), until the desired element is reached. An inefficient approach to this search, often referred as *column matching*, can substantially degrade the performance of sparse matrix factorization algorithms.

In order to quantify the costs of column matching, when an element a_{ij} is read, but its column index does not match the desired index, we refer to this as a *miss*, because an entry is read but not used. Table 2-1 summarizes the number of *gaxpys* and misses for a number of test matrices derived from the circuit simulation of real VLSI circuits. *dram* is a 2806×2806

Algorithm 2.1 (Modified Source-row Directed Form of Crout's Algorithm).

```
$k = 1$
while $k \le n$ {                                   /* LU Decomposition */
    $a_{kk}^{-1} = \frac{1}{a_{kk}}$
    foreach $j > k$ such that $a_{kj} \ne 0$
        $a_{kj} = a_{kk}^{-1} \times a_{kj}$                       /* Normalize */
    foreach $i > k$ such that $a_{ik} \ne 0$
        foreach $j > k$ such that $a_{kj} \ne 0$
            $a_{ij}^{*} = a_{ij}^{*} - a_{ik} \times a_{kj}$                /* Update */
    $k = k + 1$
    }
$k = 1$
while $k \le n$ {                                   /* Forward Elimination */
    $y_k = b_k^{(m)}$
    foreach $j < k$ such that $a_{kj} \ne 0$
        $y_k = y_k - a_{kj} \times y_j$
    $y_k = a_{kk}^{-1} \times y_k$
    $k = k + 1$
    }
$k = n$
while $k \ge 1$ {                                   /* Back Substitution */
    $x_k = y_k$
    foreach $j > k$ such that $a_{kj} \ne 0$
        $x_k = x_k - a_{kj} \times x_j$
    $k = k + 1$
    }
```

matrix representing a subsection of a dynamic memory chip. *feb* is part of a bitonic sorter chip. *mesh* represents the solution of a two-dimensional Poisson problem on a 32×32 grid. The other matrices are derived from digital signal processing circuits. The table also lists, in the sixth column, the percentage of wasted reads. In extreme cases, the percentage of these misses can be as high as 99%. In other words, if the column matching is performed as suggested in the previous paragraph, the decomposition time would be a hundred times slower than necessary. It should be noted that in earlier machines, read overhead was small compared with the time required to perform a floating point operation, making misses less significant. For this reason, older versions of circuit simulators like SPICE [Nagel75] used only a orthogonal linked list storage. However, in modern workstations and supercomputers, the overhead is much larger than the time necessary for a floating point operation, making avoiding misses much more important.

Next section will describe the Scatter-Gather approach, which provides an elegant solution

matrix	n	*nonzeros*	*gaxpys*	misses	misses(%)	total
dram	2,806	55,706	908,129	3,081,412	77.24	3,989,541
feb	10,060	136,486	291,230	26,271,225	98.90	26,562,455
mesh	961	28,881	325,920	76,604	19.03	402,524
iir12	7,310	153,858	1,734,954	1,525,921	46.79	3,260,875
iir123	11,014	229,004	2,449,967	2,522,306	50.73	4,972,273
omega	4,212	48,850	53,178	6,431,113	99.18	6,484,291
mfr	5,496	93,826	378,976	1,280,927	77.16	1,659,903
total	—	—	6,142,354	41,189,508	87.03	47,331,862

Table 2-1: Percentage of misses using column matching for sparse matrix decomposition

to the column match problem.

2.2 The YSMP Scatter-Gather Approach

The Scatter-Gather approach was first introduced in the Yale Sparse Matrix Package [Eisenstat77]. The basic idea consists of adding to the sparse matrix representation an extra vector S of size n which holds the values of a^*_{ij} in Algorithm 2.1 in scattered form. Instead of traversing a linked list to match the proper column index, the value of a^*_{ij} is directly obtained as $a^*_{ij} = \mathsf{S}[j]$, using indirect addressing. In order to obtain this representation, the i-th row is first scattered in the proper positions of the vector S, the row operation performed, and then the i-the row is gathered back into its previous compact representation.

Using the approach just described, the great advantages introduced by the elimination of the linked list traversal would be entirely offset by the overhead costs due to the scattering and gathering of a row before and after each inner loop of the update. A much better approach would be to move the scattering and gathering process out of the update loop, in which case each row would be scattered and gathered only once during the factorization. In order to achieve this goal, it is necessary to fundamentally change Algorithm 2.1 to a *Target-row directed* form of the sparse matrix decomposition.

Figure 2-2(a) depicts the *Source-row directed* nature of the Algorithm 2.1. In particular, the first execution of the update loop is highlighted, showing the first row updating rows beneath it. In general, in each update loop, a particular source row k is fixed, and used many times to update the rows beneath it.

In order to take full advantage of the Scatter-Gather mechanism, in each update double loop, a particular destination, or target row i, is fixed, and used many times to be updated by the rows above it. Figure 2-2(b) depicts the *Target-row directed* version of Crout's algorithm. In particular, the last execution of the update loop is highlighted, showing the update of the last row by other rows above it.

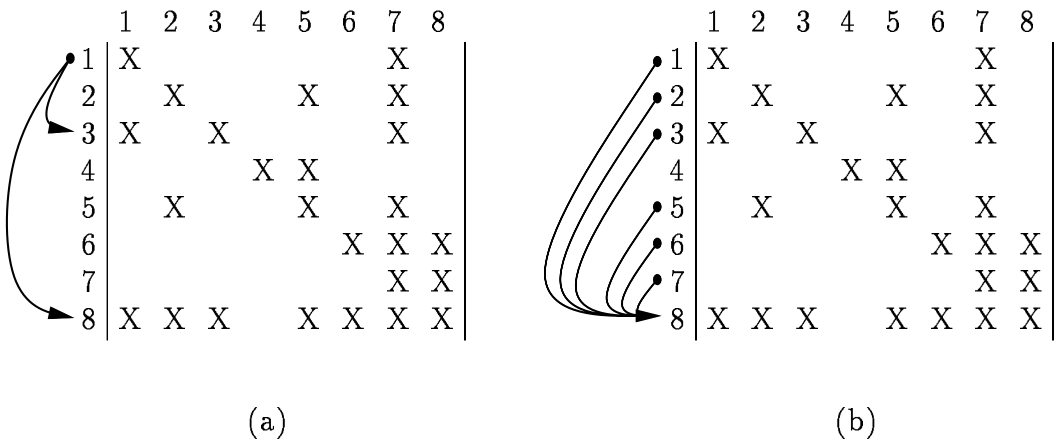


FIGURE 2-2: Source-row directed (a) and target-row directed (b) forms of sparse matrix decomposition

Given A, a non-singular sparse matrix, and a scatter vector S, Algorithm 2.2 can be used for the efficient LU decomposition of A. In order to solve $Ax = b$, the forward elimination and back substitution steps shown in Algorithm 2.1 are used without any changes. This algorithm can be slightly refined by combining the *gather* and *normalize* operations for the common elements. Even though this method is used in the actual implementation, the original form was left here for clarity. Though both algorithms represent exactly the same set of arithmetic operations, the execution in different sequential order have a great impact on the operand's access and parallelization. Later in this chapter, we shall demonstrate that both Algorithms 2.1 and 2.2 restrict the available parallelism too much.

Algorithm 2.2 (Scatter-gather Target-row Directed Form of Crout's Algorithm).

```
$i = 1$
while $i \leq n$ {                                        /* LU Decomposition */
   foreach $j$ such that $a_{ij} \neq 0$
      $S[j] = a_{ij}$                                      /* Scatter */
   foreach $k < i$ such that $a_{ik} \neq 0$
      foreach $j > k$ such that $a_{kj} \neq 0$
         $S[j] = S[j] - a_{ik} \times a_{kj}$              /* Update */
   foreach $j$ such that $a_{ij} \neq 0$
      $a_{ij} = S[j]$                                      /* Gather */
   $a_{ii}^{-1} = \frac{1}{a_{ii}}$
   foreach $j > i$ such that $a_{ij} \neq 0$
      $a_{ij} = a_{ii}^{-1} \times a_{ij}$                 /* Normalize */
   $i = i + 1$
   }
```

Figure 2-3 helps illustrate the whole Scatter-Gather process. Before each update loop, the elements of a particular target row are scattered into the proper position j in the S vector by traversing the linked row list and using the column index j as an address. This process, called *scattering*, is depicted in Figure 2-3 with arrows pointing to the scatter vector S. During the update loop, access to the source elements a_{kj} and the multipliers a_{ik} is naturally done by visiting the next element in the linked lists, just as described in Algorithm 2.1. The difference is that instead of traversing a linked list to reach a^*_{ij}, there is an immediate indirect access to S$[j]$. After all the update operations are finished, it is possible to gather back the elements of the row by traversing the linked row list and using the column index j as an address. This process, called *gathering*, is depicted in Figure 2-3 with arrows pointing from the scatter vector S back to the actual matrix elements.

The impact of scattering the destination rows has broader implications than just the simplification of the column search. Supercomputers with pipelined indirect addressing capabilities could use the Scatter-Gather algorithm and take advantage of the constant update time in order to keep the floating point arithmetic pipelines full. Another advantage of this method is that it exploits the *locality of reference* for accessing data. The destination row accesses can be done to a local cache that keeps the scatter vector S. The source row access, though accessing a large memory, is sequential, which might be an advantage for some computers.

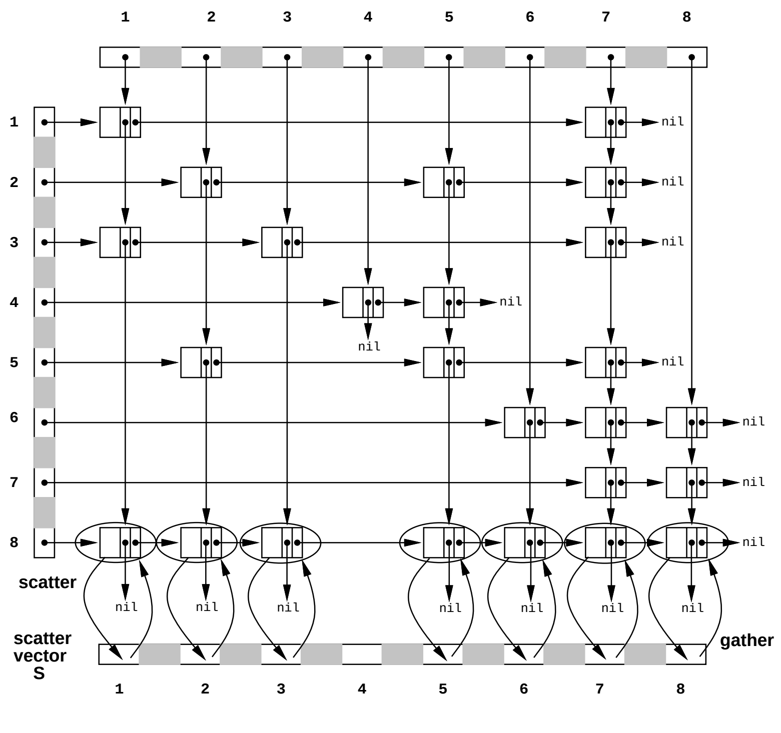


FIGURE 2-3: Scatter-Gather LU decomposition

Another advantage of the Scatter-Gather approach, also useful in all storage mechanisms that use scattering, is the *efficient interleaving*, described in detail in the next section.

2.2.1 Efficient Interleaving in Scattered Vectors

As mentioned earlier, one of the key issues for achieving high efficiency in sparse matrix computations is the design of a memory system that can keep pace with today's fastest floating point units. In order to increase the memory throughput, a possible design decision is to use faster memories, which tend to be smaller and much more expensive or, if possible, use interleaving.

Consider the memory organization shown in Figure 2-4. There are M memory modules, each one with its independent memory address registers (MAR) and memory data registers (MDR). A fast processor could issue up to M memory requests req_i during a single memory cycle if all requests are guaranteed to be directed to distinct memory modules. The address space is divided in such way that memory module i contains all addresses k such that $i = k \bmod M$. This organization is called *M-way interleaved memory*.

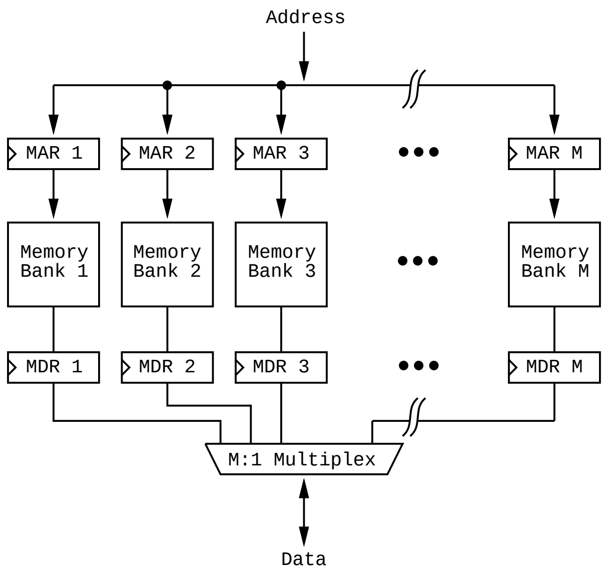


FIGURE 2-4: M-way interleaved memory system

If memory accesses were sequential, it would be easy to achieve a high degree of efficiency in interleaving, because the data would certainly be in distinct memory banks. This is the case when accessing source row elements and column indices during the update loop in Algorithm 2.2. However, the target row elements are scattered in a random order with respect to the interleaved memory banks. For example, in a two-way interleaved memory system, the *hit ratio*, or the chance of accessing a particular target row element in a given memory bank is 50%. By multiplying the number of memory modules (2) by the *hit ratio* just described (50%), we obtain the *effective usage* of the memory system. In our example, the effective usage is 1.0, or in other words, the usage of interleaved memories is a waste, as the same results would be obtained using a non-interleaved memory system.

matrix	2-way	4-way	8-way
dram	0.93	0.85	0.73
feb	0.81	0.59	0.39
mesh	0.96	0.87	0.77
iir12	0.94	0.83	0.68
iir123	0.93	0.82	0.67
omega	0.76	0.53	0.34
mfr	0.88	0.70	0.50
Average	0.88	0.74	0.58

Table 2-2: Achievable interleaving hit ratios

On the other hand, it is desirable to achieve a high degree of interleaving, specially for the memories that hold the scattered destination row. In terms of a sparse matrix solver implementation, this is particularly important because each *gaxpy* operation requires both a read and a write access cycle in the target data memory.

Fortunately, interleaving can be efficiently used for increasing the memory throughput with a small modification in Algorithm 2.2. Assuming a two-way interleaving, and using the fact that the order in which elemental updates are executed in a row update operation is irrelevant, we can pre-reorder the column indices and the source elements a_{kj} (those with $j > k$) of any particular row k in an even-odd fashion. Because the scattered structure preserves the relative position of the elements, the access to an even-numbered source element a_{kj} will be tied to the access of an even-numbered destination element $\mathsf{S}[j]$. The same holds for odd-number elements a_{kj}. By accessing the source elements in this even-odd fashion, we expect to obtain twice the throughput for the target elements as the row size increases. This concept can be extended to m-way interleaving by reordering the column indices in a round robin *module m* fashion, in order to obtain asymptotically m times the throughput.

Since the number of elements in a sparse matrix row is usually small, it is important to check the effectiveness of interleaving for the test matrices. A simple benchmark was initially devised. For each row operation, we accumulated the total number of elemental accesses and, assuming m-way interleaving, we also accumulated m times the number of elemental accesses to the most used memory bank. The ratio of the two sums above indicates the *interleaving efficiency*, which is the worst case scenario of interleaved memory access. A smart scheduling heuristic could improve slightly these figures by trying to fill in the unused memory slots at the end of one task with the appropriate accesses at the beginning of the next. On the other hand, a scheduling heuristic that cannot cope with these effects can make this figure a little worse.

Table 2-2 lists the results of the simple experiment on available interleaving efficiency for the factorization of the test matrices previously described using two-, four- and eight-way interleaving factors. On average, two-way interleaving can achieve a high *hit ratio* (90%). Four-

Matrix	No Reordering			With Reordering		
	Cycles	PE Stall	Bank Misses	Cycles	PE Stall	Bank Misses
dram	1,641,218	491,407	426,909	1,286,943	137,132	75,532
feb	1,171,617	284,448	132,139	1,106,794	219,625	84,338
mesh	637,602	203,584	145,070	503,960	69,942	23,155
iir12	3,373,011	1,055,524	802,858	2,669,872	352,385	119,309
iir123	4,868,721	1,551,858	1,152,210	3,892,886	576,023	208,670
omega	346,034	89,502	29,208	340,756	82,224	25,329
mfr	1,080,600	334,331	174,780	960,518	214,249	67,302
Total	13,118,803	4,010,654	2,863,174	10,761,729	1,651,580	603,635

Table 2-3: Effectiveness of reordering the rows with respect to column indices for interleaving

way interleaving permits on average the effective usage of three out of four memory cycles, while eight-way interleaving exhibits low efficiency. The *hit ratio* is particularly small in matrices where most rows have only 2 or 3 elements, like *feb* and *omega.*

In order to fully test the effectivity of the row reordering scheme in respect to the column indices, a detailed RTL simulation of the proposed architecture was necessary. Chapter 4 contains a detailed description of the proposed architecture. It was decided that two-way interleaved memories would be used in the processor design, since its usage provides high efficiency at a relatively low component and wiring cost. According to Table 2-2, two-way interleaving can achieve an average of $2 \times 0.88 =\ 1.76$ effective usage.

The results of a detailed RTL simulation of sparse LU decomposition for the test matrices on the proposed target architecture are summarized on Table 2-3. The data on Table 2-3 is organized to highlight the effectiveness of reordering the rows with respect to the column indices for interleaving. In the first data set, unordered matrices are used. In each data set, the first column lists the total number of clock cycles required by the sparse matrix decomposition algorithm. The second column lists the number of cycles in which the processor was stalled. The third column lists the number of interleaved memory bank misses. Each memory bank miss causes the processor to stall for one cycle. While the number of stalled cycles account for 31% of the total number of cycles, a significant majority, 71% on average (and up to 86%) of the stalled cycles were caused by interleaved memory bank misses. The exceptions are *feb* and *omega*, which have a very small number of elements per row, on average. The second data set, corresponding to matrices that have their column indices reordered for improving interleaving efficiency, exhibits consistently better results. Even though the change was not significant for matrices like *omega* and *feb*, the number of interleaved memory bank misses in the second data set are typically one-fifth of the original number of misses. The impact on processor performance is quite significant, as the number of stalled cycles was reduced to a third of the original figure, resulting in global improvements in the order of 25%.

Next section describes a special purpose single processor architecture that could take advantage of the characteristics of Algorithm 2.2 in order to achieve high utilization of the floating point unit.

2.2.2 A Scatter-Gather Special Purpose Processor

Given the current technology, the most efficient way of keeping large amounts of data, required in the storage of large sparse matrices, with high speed random access is the usage of dynamic random access memories (DRAMs). In comparison, static memories (SRAMs) are faster, but their density is much lower, which makes them unsuited for the storage of large amounts of data. SRAMs can be used to hold small amounts of temporary data, such as source rows in compact form, or up to $O(n)$ elements used to hold a small number of scattered rows. Therefore, the sparse matrix storage must be restricted to DRAMs for a reasonable implementation, while SRAM's can be used for small scale data storage.

An important aspect of DRAMs is that data access time depends on access locality. One can think of a DRAM chip as a large array of $w \times w$ elements. The access to elements in the same column is reasonably fast, while the access to elements in different columns is usually four times slower. Fast access to elements in a given column is referred to as *fast static column mode* or *fast page mode.*

Comparing the highest density memories available today, SRAMs present more than twice the speed of the DRAMs' fast static column mode access, while they have only one-sixteenth of DRAMs' density.

Figure 2-5 depicts a row-update operation being executed in a processor with a dedicated datapath for source-target alignment. In order to simplify the overall picture, the memories are not interleaved. We assume a target-row directed form algorithm with Scatter-Gather storage. The entire sparse matrix is stored in a row-ordered compact form in the DRAM, which can be part of the actual implementation of the orthogonal linked list storage. Each entry consists of numerical data (64bits) and a column index (32bits). The scattered target row i is kept in the SRAM. There is a counter that scans sequentially the elements of a row in DRAM and also a dedicated path that reads in column indices from DRAM, providing the addresses of matching data to the SRAM. Since each SRAM address is used twice, once for read and once for write, there is a shift register with a number of stages that matches the number of pipeline stages in the memory to floating point unit path, providing the write address for the SRAM aligned with the FPU output. Finally, the controller has to keep track of the number of operations executed and perform a conditional jump to stay in the update loop.

The update operation proceeds in a pipelined fashion. The pair $\{n,\ a_{kn}\ \}$ is read from the DRAM, while the previous values read are stored in the multiplier input (a_{km}) and in the SRAM read address register(m). The proper matching data (a_{im}) from the scattered vector i, is being read from the SRAM. The result of the previous multiplication $(a_{ik} \times a_{kl})$ is stored in the ALU input, being subtracted from the last data read from the SRAM (a_{il}). The result of the previous ALU operation ($a'_{ij} = a_{ij} - a_{ik} \times a_{kj}$) is written back to the proper target location (j). Note that in each clock cycle, approximately nine operations are being executed — two integer operations, two floating point operations, four memory accesses and a conditional jump.

It is important to note that this is a balanced system. During one read cycle of the DRAM, the SRAM is accessed twice, once for a read operation and once for a write operation. This is consistent with our assumption that the SRAM can operate at least at twice the speed of the DRAM's fast static column mode. In this case, we can assume that during most of the time, the DRAM is operating in fast static column mode because the accessed data is in consecutive memory positions.

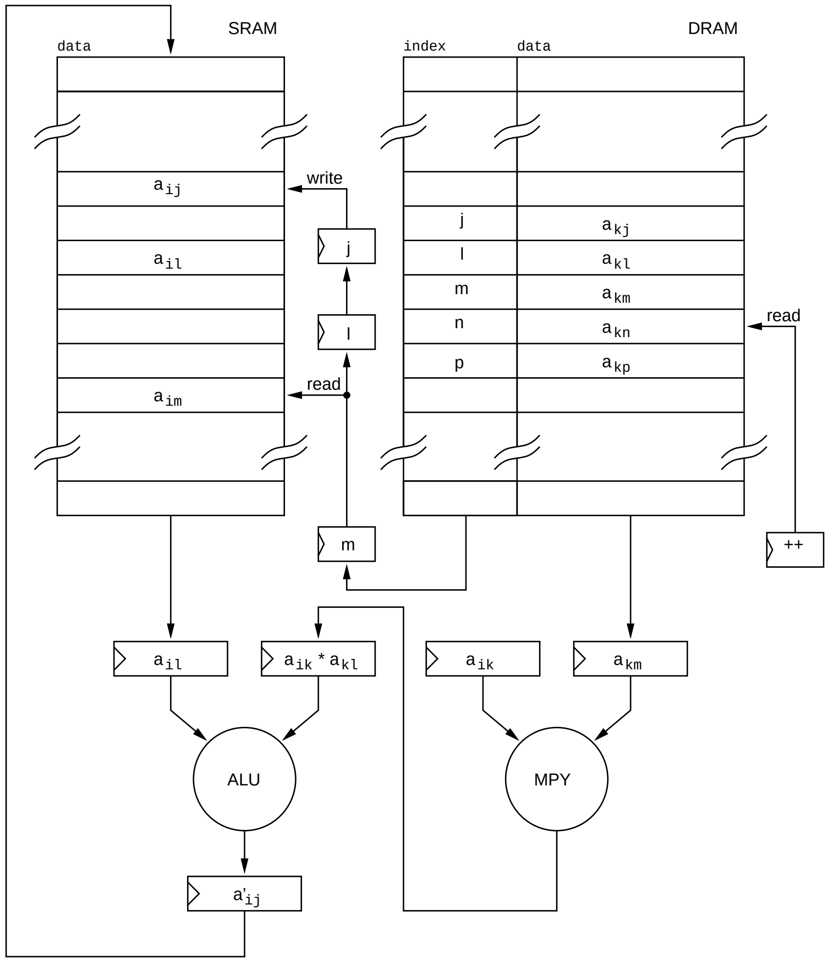


FIGURE 2-5: Dedicated datapath for source-destination column matching

A small problem with this scheme is the time spent during a row scatter, when no useful numerical operation is executed. Note that gathering can be done concurrently with normalization. However, the major problem with this technique is its extension to a multiprocessor environment. The target-row directed form restricts the available parallelism too much because it fundamentally restricts the scheduler freedom, as we shall demonstrate in Section 2.4.

The next section address the methodology used to schedule the sparse matrix factorization operations in a multiprocessor environment to achieve a high degree of utilization of the available computational resources.

2.3 Scheduling Heuristics for Fast Sparse Matrix Decomposition

Given a series of operations and their dependencies, the objective of *scheduling* consists of finding the sequence of operations for each processor in order to minimize the total completion time. It is also required that a schedule have the deadlock free property. We are interested in scheduling heuristics for a bounded number of resources, in which an optimal solution is shown to be NP-complete.

Several heuristics have been proposed for scheduling the sparse matrix factorization in a multiprocessor system. They have been briefly discussed in Chapter 1. In the following we shall discuss in detail the approach proposed by this PhD thesis. We start by providing some background information in order to better understand the *remaining completion time* heuristic. Next, we describe in detail the proposed algorithms and compare them with results from previous works. One important characteristic of our algorithm is that it can be easily tailored for different architectures, enabling us to *predict* the performance of a particular system configuration. In Section 2.6, we will discuss briefly the predictions we have used to fine tune the architecture proposed in Section 2.5.2, and compare the performance predicted by the scheduler with some actual measurements performed in a specially designed parallel computer and in a workstation.

For clarity, the following scheduling analysis will only be done for the LU decomposition. In the actual scheduler implementation, the tasks corresponding to the forward elimination and back substitution were also taken in consideration.

2.3.1 Background

Let us consider the direct solution of the system $Ax = b$, using the Modified Crout's Algorithm described in Section 2.1. Our objective is to partition and schedule amongst P processors the operations executed in the /* Update */ and /* Normalize */ loops in Algorithm 2.1 in order to minimize the time spent in a multiprocessor for performing a sparse matrix decomposition. As discussed in Chapter 1, the scheduling can be done at different granularity levels. In face of the tradeoffs between the amount of available concurrency and the required storage, we decided to use the medium-grained approach, where each pre-scheduled task represent a single row-wise operation from Algorithm 2.1. Another reason for choosing the medium-grained tasks is then the fine-grain concurrency can be exploited in a pipelined floating point unit. Using this model, the LU decomposition consists of a series of operations that follow in one of the two categories: A row-wise *normalization*, denoted by Nk, and defined by:

$$a_{kj} = a_{kj} \times \frac{1}{a_{kk}} \quad \forall a_{kj} \mid a_{kj} \neq 0 \text{ and } j > k \tag{2.2}$$

and a row-wise *update*, denoted by $k \to i$, corresponding to the row i being updated by row k, which is defined by:

$$a_{ij} = a_{ij} - a_{ik} \times a_{kj} \quad \forall a_{kj} \mid a_{kj} \neq 0 \text{ and } j > k \tag{2.3}$$

Each non-zero element a_{ik} with $i > k$ defines a row-wise update task $k \to i$. In order to keep the update naming consistent with the previously described convention, we shall refer to k as a *source row* and i as a *target row* or *destination row* of an update operation. Given an $n \times n$ sparse matrix A, n row-wise normalizations Nk, and roughly $\frac{nonzeros}{2}$ row-wise updates $k \to i$ are necessary to complete the LU factorization. This estimate assumes there are roughly as many non-zero elements in the upper and lower triangular parts of A.

The update and normalize operations just described can be executed in any sequence, provided that the following rules are respected:

i. Nk cannot be started unless all update operations that modify a_{kk} are completed.

ii. $k \to i$ cannot start before all updates that modify a_{ik} have been completed and

iii. Element a_{ij} in $k \to i$ cannot be updated before a_{kj} has been normalized.

Rules (i) and (ii) establish dependencies in the *strong* sense with respect to row-wise operations, i.e. one operation cannot start fetching operands before the other has finished storing all results, which usually is a bad characteristic for pipelined systems. Rule (iii), however, permits a row update task to start as soon as the first result of the normalize operation is available, provided that both pipelines run at the same speed. This rule establishes a new type of dependency which we shall refer to as a *mild* dependency.

These tasks and the dependencies among them could be conveniently expressed as a generalized multitask graph $G(V, E^s, E^m)$, consisting of a node set V and two arc sets E^s and E^m (corresponding to both strong and mild dependencies). This task graph has no cycles and is sometimes referred as a Direct Acyclic Graph (DAG).

In order to properly exploit pipelining in a multiprocessor vector machine we suggest the following task model:

Each node $v_i \in V$ corresponds to a pipelined task, as shown in Figure 2-6. Figure 2-6(a) shows the representation of the node v_i in the task graph. Figure 2-6(b) depicts the execution of the task in a real processor. The first piece of data enters the pipeline at $t = t_{if}$. After a time delay p_i ($t = t_{of} = t_{if} + p_i$) corresponding to the pipeline latency, the result of the first elemental operation is available. After a time c_i ($t = t_{il} = t_{if} + c_i$), which corresponds to the

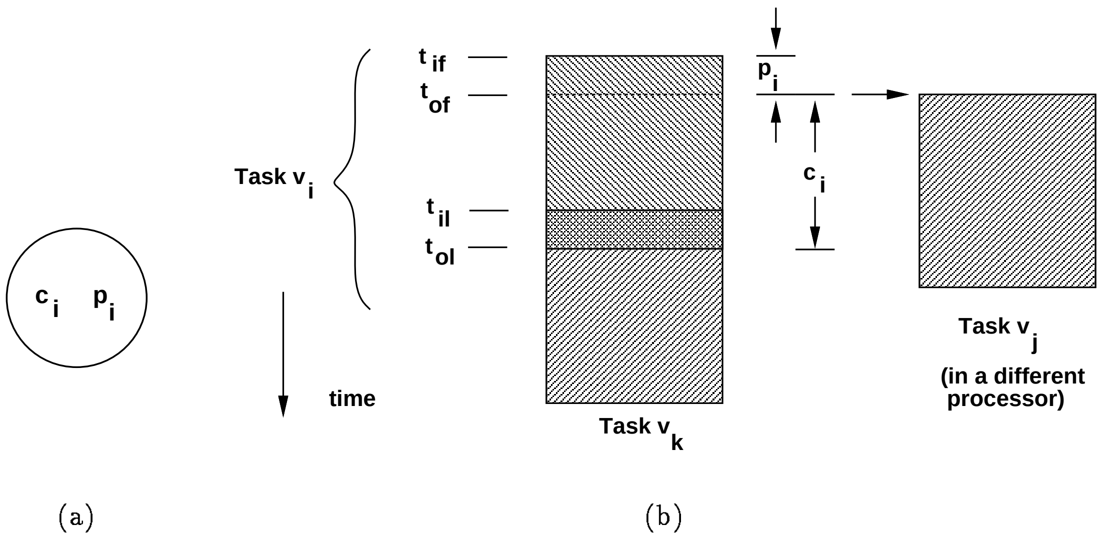


FIGURE 2-6: Task representation and its timing

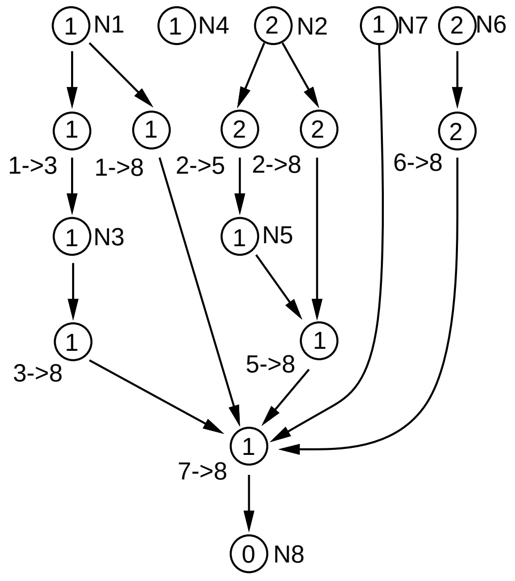
pipeline throughput times the number of elemental operations, the last piece of data is fed into the pipeline and the processor is ready to start feeding the pipeline some data corresponding to a new task. At $t = t_{ol} = t_{if} + c_i + p_i$, the result of the last elemental operation is ready, and task v_i is finished. If a task v_j depends *strongly* on v_i, it can only start at $t = t_{ol}$, but in the case of a *mild* dependency, like the situation depicted in Figure 2-6(b), it can start in another processor as early as $t = t_{of}$. If a task v_k is executed in the same processor and is either independent of or depends mildly on v_i, it can start execution as early as $t = t_{il}$. Strong dependencies are represented in the task graph by a solid arc e^s_{ij}, while mild ones are represented by a dotted arc e^m_{ij}. The ability to exploit mild dependencies plays a key role in the performance of state-of-the-art computers with deeply pipelined floating point units.

Figure 2-7(b) shows the task graph that represents the LU Decomposition of the sparse matrix depicted in Figure 2-7(a). For the sake of simplicity, the costs shown in the task graph vertices are the number of elemental update or normalize operations necessary to complete it. Ignoring pipelining for the while being, the p_i costs vanish and all dependencies become strong.

A task is called an *initial* or α-task if it does not depend on any previous operations, and is called a *terminal* or Ω-task if no task depends on it. We say that v_i *depends* on v_j if and only if there is at least one directed path $R^{j,i}_k$ that starts at v_j and finishes at v_i. Every task v_i has an associate set of *dependency paths* $R^{\alpha,i}$. Each element $R^{\alpha,i}_k \in R^{\alpha,i}$ is itself a set of vertices v_j that are traversed in the path that connects v_i to any α-node (and includes them). Furthermore, we can associate with each set $R^{\alpha,i}_k$ an integer function $S(R^{\alpha,i}_k)$ that represents the cardinality of $R^{\alpha,i}_k$. A task v_i is said to be at *level* h_i if among all the paths $R^{\alpha,i}_k$ connecting the initial tasks to v_i, the largest one contains h_i elements, or

$$
\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & X & & & & & & X & \\
2 & & X & & & X & & X & \\
3 & X & & X & & & & X & \\
4 & & & & X & X & & & \\
5 & & X & & & X & & X & \\
6 & & & & & & X & X & X \\
7 & & & & & & & X & X \\
8 & X & X & X & & X & X & X & X
\end{array}
$$

(a)



(b)

FIGURE 2-7: A sparse matrix and its associated task graph

$$h_i \;=\; \max_k \left\{ S(R_k^{\alpha,i}) \right\} \qquad \forall R_k^{\alpha,i} \in R^{\alpha,i}. \tag{2.4}$$

In Figure 2-7, tasks at the same level are grouped together in the same horizontal line. The *height* H of a task graph is defined as the level of the vertex at the highest level, or

$$H \;=\; \max_i\{\; h_i \;\} \quad \forall i \quad | \quad v_i \in V \;\; \textit{is} \;\; \textit{a terminal node.} \tag{2.5}$$

Clearly, the height of the task graph is associated with the minimum completion time of the matrix triangulation. Task graphs that are short and stout are expected to better exploit parallelism than ones that are tall and skinny. In the example in Figure 2-7, the height of the task graph is six. However, the level of the task is not enough to properly model some hardware features and different task sizes, so we introduce the concept of the *completion time* d_i of a given task v_i, which is defined as follows:

$$d_i = \; \max_k \left\{ \sum_{j=\alpha}^{i} \; (cost(v_j)) \right\} \qquad \forall v_j \;\; | \;\; v_j \in R_k^{\alpha,i} \tag{2.6}$$

where $R_k^{\alpha,i}$ is any path starting at an α-node and finishing at v_i. $Cost(v_j)$ can be either c_j (resource cost), p_j (pipeline latency time) or even $c_j + p_j$, depending on the type of the dependencies on v_j that occur while traversing the path $R_k^{\alpha,i}$. The *earliest completion time* D for the decomposition is defined as the completion time of the node with the largest d_i, or

$$D \; = \; \max_i \{ \; d_i \; \} \quad \forall i \quad | \quad v_i \in V \;\; is \;\; a \; terminal \; node. \tag{2.7}$$

The earliest completion time establishes the first barrier on the available concurrency for a given matrix and cost function. Given t_1 the sequential execution time and P processors, one could define η_D, the *maximum achievable processor utilization due to the critical path* as:

$$\eta_D \; = \; \min \left\{ \; 1, \; \frac{t_1}{D \cdot P} \right\} \tag{2.8}$$

We can also define a *critical path* $R_c^{\alpha,\Omega}$ as any path whose total cost $d_\Omega = D$. For example, in Figure 2-7, the critical path is the one containing $N2$, $2 \rightarrow 5$, $N5$, $5 \rightarrow 8$, $7 \rightarrow 8$ and $N8$, corresponding to $D = 7$.

The critical path is not a very strong bound, since we have shown in previous work that the partitioning bound is usually more restrictive than the critical path bound for row-wise operations [Telichevesky91a]. The main results of that work are reproduced in the next section. Nevertheless, the critical path plays an important role in the scheduling algorithm, as it seems natural that tasks that belong to the critical path should be executed before other less important tasks. The last definition we need for our scheduling algorithm is the *remaining completion time* δ_i of a given task v_i. Assuming v_j is a terminal task, or a Ω-task with $d_i = \; D$, and there is at least one path $R_k^{i,j}$, we define δ_i as:

$$\delta_i = \; \max_k \left\{ \sum_{x=i}^{j} (cost(v_x)) \right\} \qquad \forall v_x \;\; | \; v_x \in R_k^{i,j} \;\; \text{and} \;\; v_x \neq v_i \tag{2.9}$$

The remaining completion time is a measure of how important the early execution of a given task is, as it is known that once the task v_i is finished, the minimum time for completion of all the tasks that depend upon v_i is δ_i. Clearly δ for Ω-tasks is zero. In Section 2.3.3, we shall describe an algorithm for computing the remaining completion time for the entire task graph in linear time and a quasi-linear time scheduling scheme based on δ-heuristics.

2.3.2 Partitioning Schemes

Many heuristics have been proposed in the literature for the partitioning problem [Gerasoulis90]. However, due to the large number of rows in a matrix, usually many thousands, we could not afford to have computational costs that do not exhibit nearly linear time complexity. We focused our attention on simple algorithms that yield suboptimal results at very small computational cost.

All three partitioning methods described in [Telichevesky91a] are row-based. Once a particular row has been assigned to a particular processor, the processor will become responsible for all the update tasks that use the row as a target, as well as its normalization. If necessary, the processor will also send the normalized row to other processors to update other rows. This heuristic minimizes the amount of data transmitted in the network, as it sets the upper bound to $O(nonzeros)$.

In order to compare the relative merits of different schemes, we need to define the *partition utilization* η_P, due to the partitioning constraints, as:

$$\eta_P = \frac{t_1}{t_P \cdot P} \tag{2.10}$$

where t_P represents the barrier to the amount of parallelism available during the LU decomposition, and corresponds to the *minimum completion time due to partitioning.* t_1 is the sequential execution time, and t_P can be computed as:

$$t_P = \max_p \left\{ \sum_j C(v_j) \right\} \qquad v_j \in p \tag{2.11}$$

where $\sum_j C(v_j)$ represents the total execution time of the tasks v_j that are executed in processor p.

Fortunately, it is also possible to obtain the optimal partitioning utilization, or the *upper bound in utilization due to partitioning*, $\eta_{P,upper}$, which represents the maximum amount of parallelism available constrained by the load of a single row, and defined as:

$$\eta_{P,upper} = \min \left\{ 1, \frac{t_1}{t_{P,lower} \cdot P} \right\} \tag{2.12}$$

where $t_{P,lower}$ represents the load of the most computationally expensive row, defined as:

$$t_{P,lower} = \max_i \left\{ \sum_j C(v_j) \right\} \qquad v_j \text{ is } j \rightarrow i \text{ or} Ni \tag{2.13}$$

where $\sum_j C(v_j)$ represents the total execution time of the the tasks v_j that have row i as destination.

Three different partitioning methods were studied: the round robin $O(n)$ scheme, the element equalization $O(n \log n)$ scheme, and the load balancing $O(n \log n)$ scheme. In the round robin scheme, rows are simply assigned to processors in a *round robin* fashion. In the element equalization scheme, the rows are sorted in decreasing order of the number of elements, and then assigned in order, from the largest towards the smallest. Both these schemes severely limit the multiprocessor utilization, as shown in Table 2-4.

In the load balancing scheme, first the total cost of the execution of all tasks for each row is computed. Then, the rows are sorted in decreasing order of the row load, an $O(n \log n)$ step. Finally, the rows are processed in order, from the largest load to the smallest, and assigned to the processor p with the smallest accumulated load. As shown in Table 2-4, this scheme is nearly optimal with respect to the row-based heuristic.

matrix	P	η_P(Round Robin)	η_P(Eq. Elements)	η_P(Ld. Balance)	$\eta_{P,upper}$	η_D
feb	2	0.90	0.91	1.00	1.00	1.00
	4	0.74	0.76	1.00	1.00	1.00
	8	0.54	0.59	1.00	1.00	1.00
	16	0.40	0.39	0.60	0.60	1.00
	32	0.24	0.24	0.30	0.30	1.00
	64	0.14	0.14	0.15	0.15	0.58
iir12	2	0.99	1.00	1.00	1.00	1.00
	4	0.97	1.00	1.00	1.00	1.00
	8	0.92	0.95	1.00	1.00	1.00
	16	0.90	0.91	1.00	1.00	1.00
	32	0.75	0.80	1.00	1.00	1.00
	64	0.64	0.68	1.00	1.00	1.00
mfr	2	0.98	0.98	1.00	1.00	1.00
	4	0.89	0.92	1.00	1.00	1.00
	8	0.84	0.80	1.00	1.00	1.00
	16	0.65	0.65	1.00	1.00	1.00
	32	0.48	0.48	0.85	0.85	1.00
	64	0.31	0.32	0.42	0.43	1.00
omega	2	0.88	0.86	1.00	1.00	1,00
	4	0.63	0.67	1.00	1.00	1.00
	8	0.40	0.46	0.74	0.75	1.00
	16	0.24	0.29	0.37	0.37	1.00
	32	0.17	0.16	0.18	0.19	0.56
	64	0.09	0.09	0.09	0.09	0.28

Table 2-4: Comparison of different partitioning schemes

Table 2-4 summarizes the results for different partitioning schemes. The third and fourth columns list respectively the results for the round robin and element-based approaches, while the fifth column lists the results for the load-balancing scheme, clearly indicating its superiority over the other methods. The sixth column lists $\eta_{P,upper}$, the upper bound in utilization due to the partitioning. The seventh column lists the η_D, the maximum achievable efficiency due to critical path constraints. It is an interesting result that in all the test cases studied, the maximum utilization is limited by the partitioning scheme, and not by the critical path.

2.3.3 Scheduling Schemes

In a multiprocessor environment, there are two major factors that heavily influence the utilization of computational resources during the sparse matrix factorization. The first is the allocation of data to the processors, and the second is the task scheduling within each processor and in the communication network. In the previous section we have studied several partitioning algorithms and empirically demonstrated that even a simple balancing scheme that assigns tasks in decreasing order of its cost to the processor with smallest load yields nearly optimal partitions with respect to overall efficiency However, a simple greedy scheduling mechanism causes a significant gap between the simulated results and the theoretical maximum performance imposed by the partitioning algorithm. In this thesis we shall introduce more efficient, though fast, scheduling schemes in order to reduce this gap.

Greedy scheduling is not very efficient because it lacks knowledge about the future, as it picks enabled tasks in *any* order, and schedules them for execution. As a result of this policy, the execution of a task in the critical path could be deferred, and all processors waiting for the data output from that task will have to stall, dropping the system performance. In order to improve the utilization of the system, a good scheduling algorithm should be able to *choose* among the enabled tasks the most critical, and schedule it to be executed first.

A possible heuristic is based on the remaining completion time (δ) concept, introduced in Section 2.3.1. In this scheme, tasks with large δ are scheduled to be executed before the others, as tasks in the critical path or in heavier branches tend to have a larger δ than others. The net effect of this scheduling is not as good as the critical path scheduling described in [Sarkar89], but it has a much smaller cost, as δ can be computed for the entire task graph in linear time. Even if the efficiency obtained by the usage of this scheme is not optimal, it usually performs better than the greedy scheduling, and in some cases even approaches the bounds imposed by partitioning or by the earliest completion time D.

Critical path analysis is the basis for many scheduling heuristics previously developed. Foremost among these are PERT [Malcolm59], and CPM [Kelley61], and many other [Jackson55, Hu61, Efe82, Kim88]. Graham et al. [Graham79] studied the scheduling problem in great depth. They classify the problem of scheduling tasks to a multiprocessor with precedence relations in order to achieve minimum completion time as $P|prec|C_{max}$, and present several bounds comparing the worst case results from several algorithms with the optimal parallel execution time.

Algorithm 2.3, a variation on single source shortest path, computes the remaining completion time δ for the entire task graph in linear $O(V) + O(E)$ time. We assume that in the beginning of the algorithm the task graph is levelized and each node v_i contains a list of tasks v_j on which it depends, corresponding to the arcs in the opposite direction as shown in Figure 2-7(b). We also assume $D = 0$, and initially each node v_i contains $\delta_i = 0$.

At the end of the algorithm execution, all nodes v_i will be set to δ_i and D will be set to the *earliest completion time.* It is possible to add code /* optionally store the critical edge e^c_j */ to store the edge that caused δ_j to be maximum as the *critical edge* e^c_j. This set of critical edges form the critical path and its branches.

After the values of δ_i are found, the actual scheduling task is very simple. Assuming P processors are available, the tasks can be distributed to the processors into P bins labeled X_p, using any of the heuristics described in Section 2.3.2; we then proceed to Algorithm 2.4. We

Algorithm 2.3 (Computing Remaining Completion Time in $O(V) + O(E)$).

```
level = H
while level > 0 {
    foreach v_i such that h_i == level {
        $\delta_s = \delta_i + c_i + p_i$
        foreach v_j such that $\exists e^s_{ji}$                  /*strong nodes */
            $\delta_j = \max(\delta_j, \delta_s)$
            /* optionally store the critical edge $e^c_j$ */
        foreach v_j such that $\exists e^m_{ji}$ {                /* mild nodes */
            $\delta_m = \delta_s - c_j$
            $\delta_j = \max(\delta_m, \delta_i, \delta_j)$
            /* optionally store the critical edge $e^c_j$ */
            }
        $D = \max(D, \delta_s)$
        }
    level = level - 1
    }
```

associate with each task v_i a timestamp t_i, representing the time the task will start, and to each processor p a timestamp t_p, which indicates when the processor is free to start a new task. In the beginning of the execution, all time values are set to zero. We also associate with each task v_i the number of dependencies nd_i that must be satisfied before the task is initiated. After the Algorithm 2.4 is finished, the auxiliary variable *last* will contain the total time necessary to complete the decomposition.

The actual scheduling is executed in the **SORT** step. This scheduling is so simple that one might ask if it is correct, as it is not obvious that it will generate deadlock-free code. The following proof, by contradiction, provides necessary and sufficient conditions for a deadlock-free code:

Theorem 2.1. The scheduling generated by sorting in each processor the tasks in the decreasing order of δ is deadlock-free.

Proof. This proof is divided into three parts, the first corresponding to the uniprocessor case and the remaining regarding interprocessor deadlocks.

(a) Suppose two tasks v_i and v_j are in the same processor and v_i depends on v_j. A deadlock can occur if and only if v_i is scheduled to be executed before v_j. However, if v_i depends on v_j there must be at least one directed path $R^{j,i}_k$ and therefore $\delta_j = \delta_i + \sum_{k=j}^{i} cost(v_p)$ provided $v_p \in R^{j,i}_k$ and $v_p \neq v_j$. Assuming $cost(v_p) > 0$, otherwise not physically possible, $\delta_j > \delta_i$ and v_j will be scheduled to be executed before v_i, which contradicts the initial deadlock hypothesis.

Algorithm 2.4 (δ-Based $O(V \log V)$ Scheduling Algorithm).

```
$p = 1$
while $p \leq P$ {
    SORT all tasks $v_i \in X_p$ in decreasing order of $\delta_i$
    $p = p + 1$
    }
while $\bigcup X_p \neq \emptyset$ {
    $p = 1$
    while $p \leq P$ {
        visit the next ordered task to be scheduled $v_i \in X_p$
        if $(nd_i == 0)$ { /* optionally insert greedy algorithm */
            $t_i = t_{if} = \max(t_i, t_p)$
            $t_{ol} = t_{if} + c_i + p_i$
            $t_{of} = t_{if} + p_i$
            $t_p = t_{il} = t_{if} + c_i$
            foreach $v_j$ such that $\exists e^s_{ij}$ {                    /* strong */
                $nd_j = nd_j - 1$
                $t_j = \max(t_j, t_{ol})$
                }
            foreach $v_j$ such that $\exists e^m_{ij}$ {                      /* mild */
                $nd_j = nd_j - 1$
                $t_j = \max(t_j, t_{of})$
                }
            $last = \max(last, t_{ol})$
            $X_p = X_p - v_i$
            }
        $p = p + 1$
        }
    }
```

(b) Suppose v_i and v_j are in a processor p_r and v_m is in processor p_s. An interprocessor deadlock can occur if and only if v_m depends on v_j and v_i depends on v_m, but v_i is scheduled to happen before v_j. Using the same arguments as in part (a) we can say $\delta_j > \delta_m$ and $\delta_m > \delta_i$. Using associativity $\delta_j > \delta_i$, which finishes the proof of the theorem by the same arguments as in (a).

(c) By induction on (b) we can prove that there will be no deadlocks in any number of processors. □

As we shall see in Section 2.3.4, the code generated by the δ-based scheduling does not produce very good results in terms of efficiency. A simple modification on the plain δ-based scheduling algorithm yields much better results. One can imagine that δ-heuristic and greediness are conflicting issues. While the δ-heuristic blindly waits for the most critical task to be scheduled even if it might execute another less important task to *fill up* the available time slots, the greedy heuristic would just schedule anything available, regardless of the fact that sacrificing a little time waiting for a more important task could actually reduce the total time. It is clear that during the scheduling, if a less important enabled task could fill up some available time slots without delaying the most critical task chosen by the δ heuristic it should be scheduled to execute first. However, if the insertion would delay the execution of the critical task, it is necessary to have some heuristic to decide whether or not it is worth sacrificing the free time slots. The easiest way to manage this situation is to establish quantitatively a *scheduling elasticity* κ, which is discussed in the following.

Algorithm 2.5 (Added code for the δ-Greedy Algorithm).

```
if ($t_i \leq t_p$)                                              /* no slack */
    $t_i = t_p$
else {                                                           /* slack */
    $v_s = v_i$
    $slack_s = slack_i = t_i - t_p$
    foreach $v_j \in X_p$ such that ($nd_j$ == 0) and ($slack_s \neq 0$) { /* search */
        $slack_j = \max(t_j - t_p, 0)$
        if ($slack_j < slack_s$) and ($slack_j + c_j < \kappa \cdot slack_i$) {
            $slack_s = slack_j$
            $v_s = v_j$
            }
        }
    $v_i = v_s$
    $t_i = t_p + slack_s$
    }
```

The new algorithm is basically the same as Algorithm 2.4, with the exception that the line annotated by /* insert greedy algorithm */ is substituted by the Algorithm 2.5.

Immediately after the line annotated with /* insert greedy algorithm */ in Algorithm 2.4, we set $t_{if} = \max(t_i, t_p)$. This code says that v_i cannot start before it is enabled by the completion of tasks it depends upon (t_i), or before the processor is ready to execute it (t_p), whichever occurs last. If $t_i > t_p$, the processor stalls. However, there might be another task v_j such that $t_j < t_i$, but $\delta_j < \delta_i$ and $nd_j == 0$. In this case, we could execute v_j before v_i in order to reduce the slack. Figure 2-8 depicts four possible timing situations, which might occur if we tried to insert v_j before v_i.

In (a) and (c), $t_j > t_p$, and therefore there is still a little slack left, while in (b) and (d), v_j causes no slack. In (a) and (b), the insertion of v_j causes the beginning of v_i to be delayed, while in (c) and (d), there is no delay involved, since the inserted task finishes before v_i is

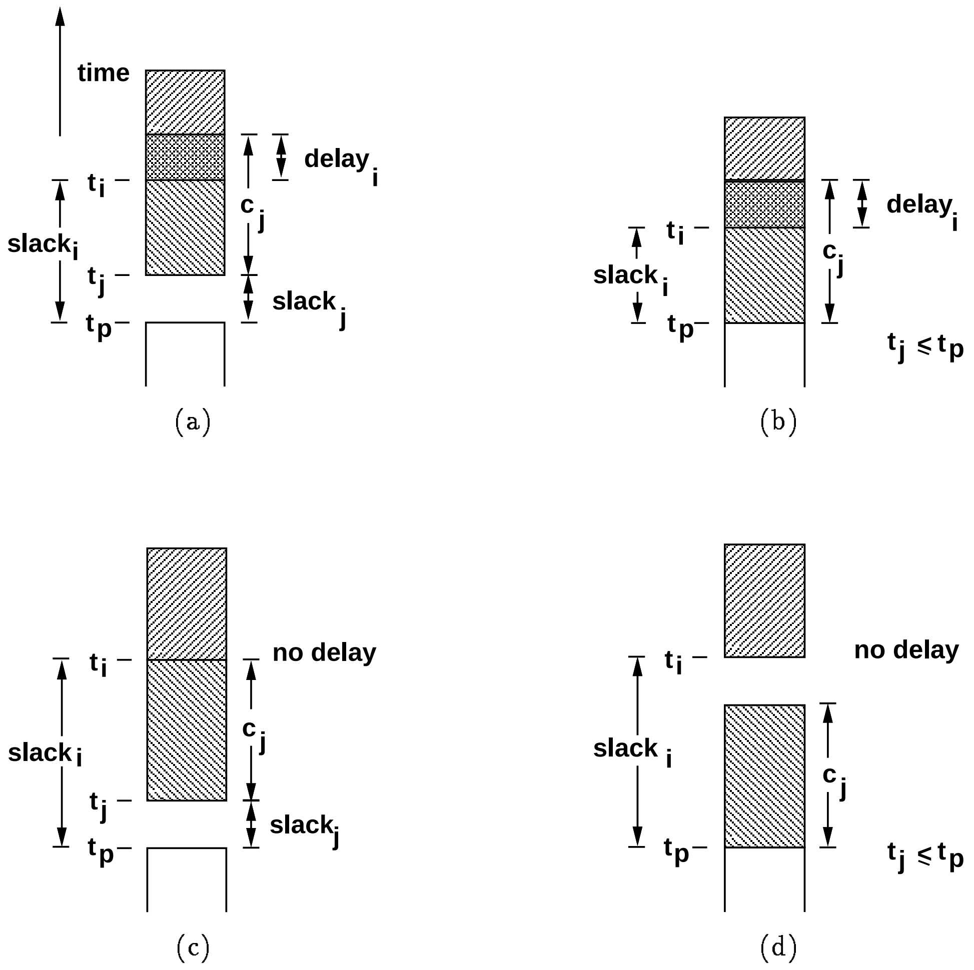


FIGURE 2-8: Inserting a task v_j before v_i

enabled. The heuristic used to decide whether or not insert a task v_j depends on κ, so if the condition

$$\frac{slack_j + c_j}{slack_i} < \kappa \tag{2.14}$$

is satisfied, we insert v_j before v_i. If $\kappa \leq 0$, no insertion is permitted, and the algorithm yields the same results as the plain δ-based scheduling. If $\kappa \leq 1$, no conflict happens between the δ-heuristic and the greedy algorithm. In this case, the insertions depicted in Figure 2-8 parts (a) and (b) are not allowed. If $\kappa > 1$, there is a compromise between the greedy algorithm and the δ-based scheme. Empirically, the algorithm results are very much insensitive to the actual value of κ, as long as $\kappa > 1$.

In the following section, we discuss the results of these scheduling heuristics and compare them with previous results.

Matrix	P	$\eta_{P,L}$	$\eta_{P,G}$	$\eta_{P,\delta}$	$\eta_{P,\delta-G}$	η_{max}
dram	2	0.94	0.99	0.99	0.99	1.00
	4	0.84	0.99	0.99	0.99	1.00
	8	0.71	0.99	0.99	0.99	1.00
	16	0.58	0.88	0.83	0.96	1.00
	32	0.44	0.79	0.76	0.89	1.00
	64	0.28	0.48	0.55	0.56	1.00
feb	2	0.88	0.91	0.85	0.95	1.00
	4	0.71	0.87	0.74	0.87	1.00
	8	0.51	0.61	0.63	0.82	1.00
	16	0.33	0.38	0.60	0.60	0.60
	32	0.19	0.22	0.30	0.30	0.30
	64	0.11	0.13	0.15	0.15	0.15
iir12	2	0.97	0.99	0.93	1.00	1.00
	4	0.91	0.99	0.85	1.00	1.00
	8	0.83	0.89	0.74	0.87	1.00
	16	0.70	0.77	0.53	0.75	1.00
	32	0.52	0.64	0.43	0.68	1.00
	64	0.37	0.50	0.32	0.56	1.00

Table 2-5: Comparison of different scheduling schemes

2.3.4 Scheduling Results

In order to properly compare the quality of the code generated by different scheduling schemes, we are interested in the processor utilization η achieved by the algorithm. We compare the *completion time* $t_{P,S}$, which is the total time required to complete the decomposition in P processors using a particular scheduling scheme S, with t_1, the sequential execution time. We can define the *utilization* $\eta_{P,S}$ achieved by the usage of a particular scheduling scheme S with P processors, by:

$$\eta_{P,S} = \frac{t_1}{t_{P,S} \cdot P} \tag{2.15}$$

It is also useful to compare $\eta_{P,S}$ with the *upper bound in utilization* η_{max}, due either to partitioning constraints or due to the critical path, and defined by:

$$\eta_{max} = \min\{\, \eta_{P,upper},\ \eta_D\} \tag{2.16}$$

where $\eta_{P,upper}$ is the upper bound in utilization due to partitioning, as defined in Section 2.3.2, and η_D is the upper bound in utilization due to the critical path, as defined in Section 2.3.1. Table 2-5 lists the scheduling efficiency for relevant test matrices previously described, and for various values of P. The second column in Table 2-5 represents P, the number of processors used in the simulation experiment. The next four columns show the processor utilizations $\eta_{P,L}$, $\eta_{P,G}$, $\eta_{P,\delta}$ and $\eta_{P,\delta-G}$ respectively for the level-based , greedy, plain δ-based (Algorithm 2.4) and δ-greedy (Algorithm 2.4 with modifications described in Algorithm 2.5) scheduling schemes. The last column in the table shows η_{max}, the upper bound in utilization.

Superiority of the greedy algorithm over the level-based approach is evidenced in all results presented. This result was expected because the level-based scheme incurs in a lot of processor idle time, as the processors are not permitted to execute the tasks at the next level until all tasks at the present level are completed. The greedy heuristic overcomes this shortcoming, but still lacks the ability to schedule critical tasks before other less relevant tasks. Compared with the δ based heuristic, the greedy heuristic is usually better, but tends to exhibit poorer performance as the number of processors grow.

The δ-greedy scheduling yields the best utilization in the majority of cases, as it combines the advantages of the greedy heuristic with the concept of remaining time for completion. The improvement that results by permitting the insertion of tasks in the processor idle periods can be observed by comparing columns for $\eta_{P,\delta}$ and $\eta_{P,\delta-G}$ in Table 2-5.

It is important to observe that in *feb*, the δ-greedy heuristic approaches the theoretical maximum value of processor utilization as the number of processors in the system increases. In the remaining two cases, *dram* and *iir12*, it falls short of the theoretical utilization at least for a moderate number of processors. The reason for this appears to be the nature of the dependencies among the tasks and the sub-optimal way in which data is assigned to different processors, which in turn reduces the overall scheduling freedom in the system.

The heuristics described in Section 2.3.3 are very simple, yet yield experimentally very good schedules. Perhaps the most distinguished feature is that they exhibit fast execution time, comparable with the actual decomposition on a workstation. The complexity of these algorithms is $O(V \log V)$, compared with $O(V^2)$ of the critical path algorithm. Since the number of tasks involved in the test matrices is on the order of hundreds of thousands, the ratio $\frac{V^2}{V \log V}$ yields tens of thousands, which in terms of CPU time represents days instead of seconds. In this case, the precompilation step itself becomes a bottleneck and its results are meaningless.

The results presented in this section are valuable for comparing different algorithms. However, the cost functions they represent are not realistic due to implementation restrictions on the previous algorithms. In Section 2.6 we use the δ-greedy heuristic with extensions for more realistic simulations.

So far, it was assumed by the scheduling algorithm that any data necessary for the task execution would be readily accessible in constant time. In other words, it was assumed that the scheduler had total freedom to pick any operation as long as they did not violate the dependency constraints. However, this assumption is not correct if the Scatter-Gather storage is used, as only a single row i can be scattered in each processor at a given time. In that case, it is necessary to schedule a scatter operation on row i, schedule all the updates to that row, normalize it, and gather it back into the dense form. Instead of having $O(nonzeros)$ tasks to choose from, the scheduler is restricted to $O(n)$ tasks, corresponding to the *coarse grain parallelism* in Sadayappan's nomenclature [Sadayappan88].

The next section discusses an efficient storage organization based on overlapped-scattered arrays (OSA), which allows *all* rows to be simultaneously scattered, thus allowing row-wise operation freedom to the scheduling mechanism, corresponding to *medium grain parallelism*.

2.4 Overlapped-Scattered Arrays for Sparse Matrix Factorization

The Overlap-Scatter Array (OSA) representation is a mapping from a two dimensional representation of a matrix onto a one dimensional vector in such a way that the distance between any two non-zero elements in the same row is preserved, and no pair of non-zero elements occupy the same *physical* position in the vector [Sadayappan88]. A memory efficient OSA organization intersperses the non-zeros of distinct rows in order to minimize the size of the resulting vector. Each row k in the original matrix is associated with an offset that represents the distance between the physical location where a_{k0} would be stored and the origin of the OSA vector.

Figure 2-9 depicts a sparse matrix and its associated OSA representation. Any non-zero element a_{ij} of the original matrix can be accessed in constant time by adding the column index j with the appropriate row offset *offset[i]*. For example, if we want to access a_{37} we simply add *7* to *offset[3]= 15* in order to obtain the proper array index, *22*. Elements 24, 25 and 27 represent wasted memory positions. In this hand packed example, the memory utilization is $\frac{25}{28} = 89\%$.

$$\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & X & & & & & & X & \\
2 & & X & & & X & & X & \\
3 & X & & X & & & & X & \\
4 & & & & X & X & & & \\
5 & & X & & & X & & X & \\
6 & & & & & & X & X & X \\
7 & & & & & & & X & X \\
8 & X & X & X & & X & X & X & X
\end{array}$$

Row	Offset
1	0
2	10
3	15
4	-2
5	21
6	13
7	6
8	3

1	2	3	4	5	6	7	8	9	10	11	12	13	14
a_{11}	a_{44}	a_{45}	a_{81}	a_{82}	a_{83}	a_{17}	a_{85}	a_{86}	a_{87}	a_{88}	a_{22}	a_{77}	a_{78}

15	16	17	18	19	20	21	22	23	24	25	26	27	28
a_{25}	a_{31}	a_{27}	a_{33}	a_{66}	a_{67}	a_{68}	a_{37}	a_{52}	●	●	a_{55}	●	a_{57}

FIGURE 2-9: Overlap-Scatter representation of a matrix

The OSA representation of a sparse matrix is also an effective solution to the source-target element matching problem during update operations. During a normalize operation, the (upper-triangular) non-zero elements a_{ij} from a source row can be stored in a temporary, dense vector along with its column indices j. During update operations the elements of this vector are accessed sequentially and the column indices are simply added with the target row offset in order to compute the address of the matching target element. All these operations are very regular and therefore can be easily vectorized in a general purpose supercomputer and are especially attractive for the special purpose hardware.

The major feature of the OSA structure is its impact on concurrency. Since all the data in the matrix is readily accessible, there is no reason to limit ourselves to a source-oriented or target-oriented algorithm, resulting in much better schedules in a multicomputer environment.

Figure 2-10 depicts the impact of different storage techniques on multiprocessor performance for the sparse decomposition of the example test matrix *dram*. If the multiprocessor scheduling is constrained in such a way that only one scattered row can be used as a target in each processor, which is the case of the Scatter-Gather algorithm, the utilization attained is so poor that it does not justify the usage of a parallel processor. On the other hand, the additional freedom given to the scheduler by the OSA structure, where all target rows are scattered, is enough to yield utilizations as high as 90% even on 32 processors.

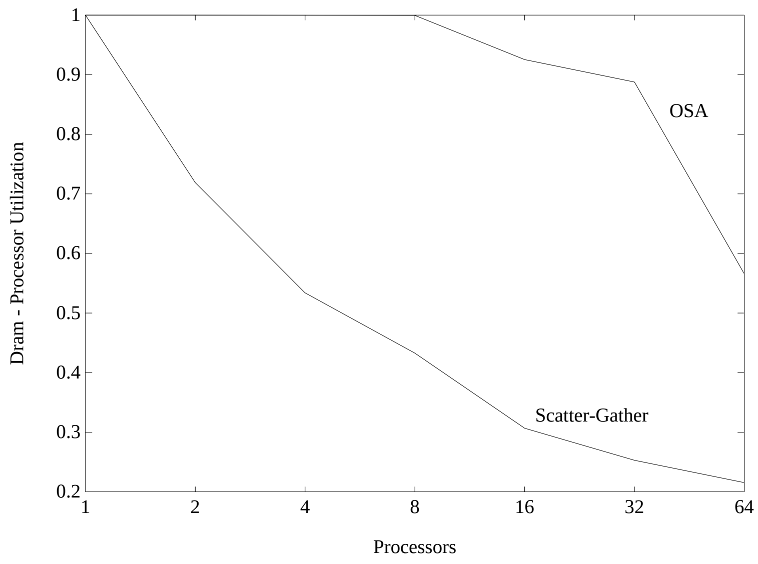


FIGURE 2-10: Simulated processor utilization for Scatter-Gather and OSA-based storage

In spite of the great advantages for multiprocessing, the OSA structure exhibits many drawbacks in terms of local processing. The most obvious ones are the wasted memory positions and the initial packing overhead, which will be empirically proven not very relevant for circuit simulation in Section 2.4.1. However, other less obvious disadvantages actually play a major

Matrix	%mem. ut.	time(OSA)	time(LU)
dram	37.49	10.800	4.922
feb	62.98	8.160	1.016
mesh	64.48	3.086	0.672
iir12	38.22	33.321	4.605
iir123	38.14	50.621	6.612
omega	75.14	1.574	0.265
mfr	53.92	8.620	0.949
Average	52.91	5.223	0.889

Table 2-6: OSA overheads

role in the local processor performance. First, the targets for the row update operations, which must be accessed *twice* per *gaxpy*, are spread exactly in a large memory which tends to be the slowest component of the system. Second, given the scheduler freedom, there is a random memory access pattern to all OSA elements, making it very difficult to exploit the *locality of reference*, which in turn makes cache utilization very ineffective. Finally, there is the issue of DRAM static column misses, which will be further discussed in Section 2.4.3, in the context of special purpose hardware for OSA. In the next section, we will discuss the OSA overhead costs.

2.4.1 OSA Overhead Costs

The most important figures of merit for evaluating the OSA overheads are: the percentage of memory utilization and the time necessary to compute the proper OSA offsets. The problem of packing the matrix into this form using a minimum amount of memory is NP-complete [Tarjan79]. Several heuristics have been proposed to quickly pack the matrix with reasonable memory utilization [Ziegler77, Tarjan79, Rao88, Sadayappan89, Trotter90b]. The discussion of these methods is beyond the scope of this thesis, so we will simply use their results to discuss if the utilization of the OSA structure is feasible.

According to experimental data in Table 2-6, both the memory utilization and packing time overhead introduced by the OSA algorithm are reasonable. Typically, half of the OSA array positions are left unused, and the packing algorithm requires no more than $10\times$ the LU decomposition time. If the system of equations has to be solved only once or only a few times, the usage of OSA arrays is not effective. However, considering that in a typical circuit simulation the matrices need to be decomposed thousands of times, the initial packing overhead cost is insignificant.

Table 2-6 summarizes the OSA memory utilization and the time spent for generating the OSA structure on a DEC 5000/200 for a number of test matrices derived from the circuit simulation of real VLSI circuits. The algorithm used is a variation on the flyback storage method (FSM) described in [Trotter90b]. The table also lists, in the last two columns, the number of floating point operations and the actual LU factorization (without counting the Markowitz reordering time) time for these matrices, using the SPICE sparse matrix package 1.3b [Kundert88]. All times are in seconds and the same optimization options were used for both programs.

There are many possibilities in exploiting the properties of the OSA structure. In the following we will discuss the implementation of a fast sparse matrix solver for workstations, which delivers, on average, twice the performance of the Sparse 1.3b package [Kundert88].

2.4.2 OSA-Based Fast Sparse Matrix Package

A very simple, yet powerful sparse matrix decomposition scheme is shown in Algorithm 2.6.

Algorithm 2.6 (OSA-Based Fast Sparse Matrix Decomposition Algorithm).

```
Generate INST[$ninst$], OSA[$\frac{nonzeros}{mem\ ut.}$], and INDEX[$nonzeros$] structures
$i = 0$
while $i < ninst$ {
    $offset =$ INST[$i$].offset
    $size =$ INST[$i$].size
    COLUMN = INST[$i$].column
    if (INST[$i$].type == 'normalize') {                    /* Normalize */

        OSA[$offset$] = $mult = \frac{1}{\mathsf{OSA}[offset]}$
        $j = 0$
        while $j < size$ {
            BUFFER[$j$] = $mult \times$ OSA[$offset$ + COLUMN[$j$]]
            $j = j + 1$
            }
        }
    else {                                                   /* Update */

        $mult =$ OSA[$offset$]
        $j = 0$
        while $j < size$ {
            OSA[$offset$ + COLUMN[$j$]] = OSA[$offset$ + COLUMN[$j$]] $-$ $mult \times$ BUFFER[$j$]
            $j = j + 1$
            }
        }
    $i = i + 1$
}
```

matrix	IBM RS6000/540			Sparcstation 2			DEC 5000/200		
	Sparse1.3b	OSA	s	Sparse1.3b	OSA	s	Sparse1.3b	OSA	s
dram	780	280	2.8	5,300	1,099	4.8	4,922	4,781	1.0
feb	340	170	2.0	867	-	-	1,016	617	1.6
mesh	180	90	2.0	567	316	1.8	672	398	1.7
iir12	1,430	670	2.1	3,433	1,567	2.2	4,605	2,430	1.9
iir123	2,030	950	2.1	5,033	2,217	2.3	6,612	3,781	1.7
omega	90	40	2.3	217	100	2.2	265	137	1.9
mfr	290	170	1.7	817	-	-	949	621	1.5
Average	274	129	2.1	742	329	2.4	889	503	1.5

Table 2-7: OSA versus Sparse 1.3b

The symbolic structure of the sparse matrix is first analyzed and its OSA representation is created. This precompilation phase also generates a sequence of *ninst* instruction records, or *task descriptors*, each representing a row-wise update or normalization, which will be executed sequentially during the decomposition.

Each instruction contains information on the type of the operation, its size, the corresponding offset of the target row in the OSA structure and the starting address of the column indices of the source row. INST is a vector that contains all these instructions sequentially. OSA is a vector containing the OSA structure. INDEX is a vector that contains the column index for all the (upper-triangular) elements of the matrix sequentially, in row-major order. Each COLUMN array is a subset of the INDEX vector. BUFFER is a temporary vector of size n that contains one source row stored in sequential order.

In order to save space and avoid unnecessary indexing operations, the column indices in the INDEX array are actually offset by the row number, so that the diagonal will have column index zero, while upper-triangular elements will have positive column indices. In order to compensate for this, the offset field of the instruction is added to the diagonal column index.

Table 2-7 lists the execution times (in mili-seconds) for matrix factorization in various architectures using the Sparse1.3b package [Kundert88] and using OSA-based Algorithm 2.6. The table also lists the speedup obtained in each platform using the OSA-based algorithm. The OSA algorithm presents roughly a factor of two speedup on average. This factor is likely to increase in vector machines because of the high potential for vectorization present in the algorithm.

In the next section we describe a pipelined special purpose system that uses the OSA structure in order to achieve high efficiency, without the heavy concurrency constraints imposed by the architecture previously described in Section 2.2.2.

2.4.3 *OSA*-Based Fast Sparse Matrix Decomposition Hardware

In attempting to design special purpose hardware that could exploit the advantages of the *OSA* structure, one immediately realizes that only using DRAMs we could effectively store a large structure such as the *OSA* vector for large matrices. However, there is a fundamental problem in keeping the *OSA* data in DRAM: the target row data for updates, which requires two accesses for each elemental operation, would be placed in the slowest component of the system.

Fortunately, it is possible to overcome this problem by interleaving multiple memory banks, as described in Section 2.2.1. The same interleaving results that apply for the Scatter-Gather approach also hold for the *OSA*-based algorithm. In fact, there is an extra degree of freedom for the *OSA*-based interleaving, depending on the value of *offset* for a given row. If the row *offset* is an even number, even-numbered accesses in the source memory will correspond to even-numbered accesses in the target memory, while odd-numbered source accesses will correspond to odd-numbered target accesses. Conversely, if the row *offset* is an odd number, even-numbered accesses in the source memory will correspond to odd-numbered accesses in the target memory, while odd-numbered accesses in the source memory will correspond to even-numbered accesses in the target memory. A clever scheduling and memory allocation scheme can take advantage of this extra degree of freedom to further increase the interleaving effectiveness.

Figure 2-11 depicts a row update operation being executed in a special purpose processor with dedicated datapath for fast *OSA* addressing. The DRAM, which holds the *OSA* data structure, is a two-way interleaved memory. The SRAM contains one (or more, depending on the scheduling algorithm) source row, with its elements stored sequentially along with their corresponding column indices (in even-odd form). The other components of the system are similar to those discussed in Section 2.2.2 with little modifications.

The other elements are: a counter that scans sequentially the SRAM addresses, a dedicated datapath, which reads in column indices from SRAM and feeds an adder to generate the proper *OSA* address, and two shift registers to store addresses for both even and odd memory write-backs, with the proper length such that their contents are aligned with the floating point unit output.

The algorithm implemented is a slight variation of Algorithm 2.6. The only modification required is that the vector COLUMN is now kept in SRAM for updates. Due to the limited space available in SRAM, it is necessary to copy the proper subset of the INDEX vector from DRAM during a row normalize.

The update operation proceeds in a pipelined fashion. The pair $\{r, a_{kr}\}$ is read from the dense source row vector in SRAM. At the same time, the element previously read, a_{kq}, is multiplied by a_{ik} — corresponding to *mult* $\times$ BUFFER$[j]$ in Algorithm 2.6, while q is added to the offset `off` — corresponding to *offset*+COLUMN$[j]$ in Algorithm 2.6. The address previously generated, `off` $+\ p$, is used to read the matching target data a_{ip} from the *even* memory bank.

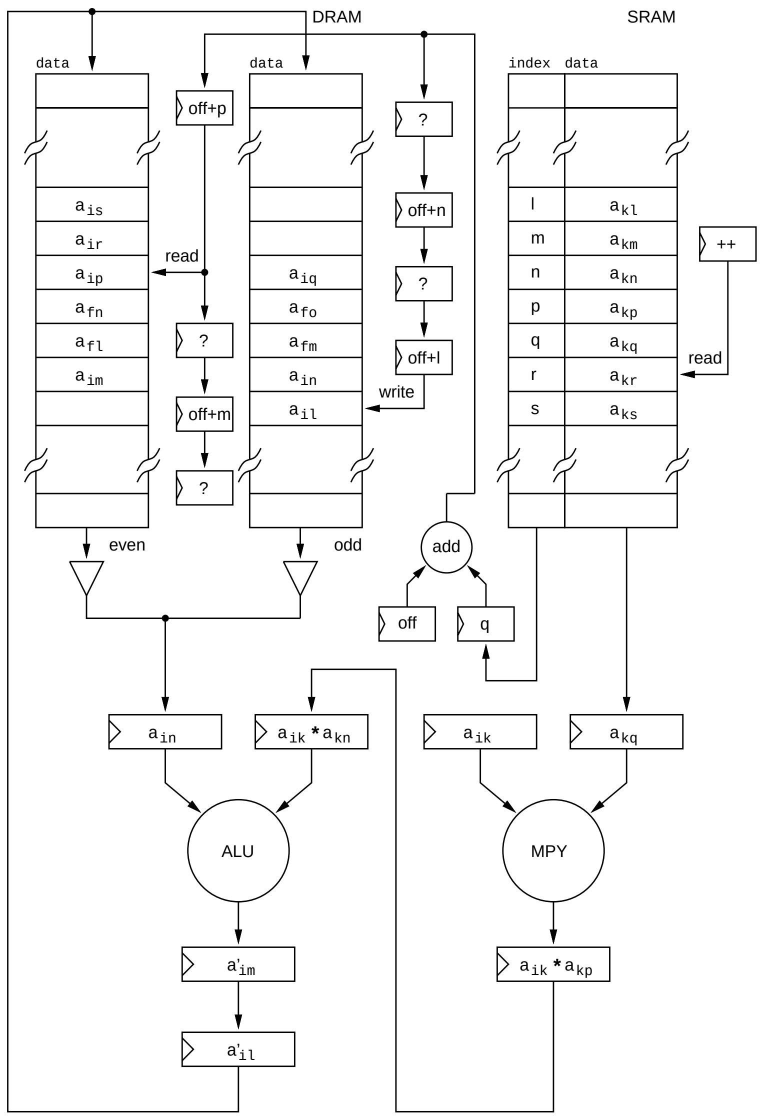


FIGURE 2-11: Dedicated datapath for fast updates with interleaved OSA access

The result of the multiplication that started two cycles before $(a_{ik} \times a_{kn})$ is subtracted from the data previously read (from the odd memory bank) a_{in} . This delay is necessary for the proper pipeline alignment. Also, it can be caused by the intrinsic characteristics of the floating point multiplier hardware, which, for this example, exhibits a latency of two clock cycles. Assuming the ALU also exhibits the same latency, its output register contains the results of the subtract operation initiated two cycles before $a'_{il} = a_{il} - a_{ik} \times a_{kl}$, which in turn are being written back to the *odd* memory bank. In addition to all these operations, the read pointer to SRAM is being incremented — corresponding to $j = j + 1$ in the algorithm, and a counter (not shown in the Figure 2-11) is being decremented to check the size of the operation and a conditional jump is taken to stay in the loop if the counter did not reach zero. Note that in each clock cycle approximately ten operations are being executed — three integer operations, two floating

Matrix	Uniform Access			Paged Access		
	Cycles	PE Stall	% Stall	Cycles	PE Stall	% Stall
dram	1,241,873	69,222	5.6%	1,523,775	351,124	23.0%
feb	970,178	89,248	9.2%	1,791,373	910,443	50.8%
mesh	483,779	25,040	5.2%	607,502	148,763	24.5%
iir12	2,490,026	112,484	4.5%	3,304,100	926,558	28.0%
iir123	3,585,329	178,524	5.0%	4,946,384	1,539,579	31.1%
omega	283,852	25,828	9.1%	519,517	261,493	50.3%
mfr	834,161	61,763	7.4%	1,300,575	528,177	40.6%
Average	699,080	47,476	5.4%	1,067,171	356,871	28.6%

Table 2-8: Influence of static column misses on the performance of the OSA-based architecture and algorithm

point operations, four memory accesses and a conditional jump.

Comparing this processor architecture for OSA with the much simpler one described in Section 2.2.2 with support for Scatter-Gather, the *gaxpy* throughput is the same, and the only time saved is the Scatter-Gather overhead, which is only $O(nonzeros)$. In addition to the previously described problems associated to the OSA structure, it is also important to mention the additional problem posed by DRAM static column misses. Since the OSA-based algorithm causes a random access to different portions of the DRAM, there is a high ratio of static column misses. Worse yet, matrices like *feb*, with very long rows ($n = 10,600$), have elements belonging to the same row spread through several distinct DRAM static column pages, which are typically 1,024 or 2,048 words long. Also, the even-odd access pattern used for increasing the interleaving efficiency seems empirically to cause more static column misses.

Table 2-8 highlights the effects of static column misses. The data was obtained by a detailed RTL simulation of the architecture shown in Figure 2-11 using a slight modification of Algorithm 2.6. In the first data set, a memory with uniform data access is assumed. This could be achieved by substituting the DRAMs in the Figure 2-11 by much more expensive, much lower density static memories. The second set, represents a memory system implemented with DRAMs, assuming a three-cycle penalty for each static column miss, and each static column 2,048 words long. In each set, the first column lists the total number of clock cycles required by the sparse matrix decomposition algorithm. The second column lists the total number of cycles in which the processor was stalled. The third column lists the percent rate of the stalled cycles relative to the total number of cycles. In the first set, stalls are only caused by interleaved memory bank misses, which are rather infrequent, on average 5.4% of the total number of cycles, thanks to the even-odd reordering scheme. On the second data set, the number of stalled cycles grows on average 7.5$\times$ or as much as 8$\times$ due to the influence of the static column misses, causing an average slowdown of 52% on the total number of cycles.

Assuming the floating point throughput can be substantially increased, as the processor speed tends to grow faster than the memory speed, one could also argue that just by adding a relatively small amount of hardware in order to make the SRAM interleaved, and reverting back to the old Scatter-Gather scheme, it would be possible to actually double the *gaxpy* throughput, and at the same time eliminate all the problems associated with the OSA structure. However, the multiprocessor utilization would be extremely poor. In the next section we will introduce the O^2SA storage scheme, which successfully bridges the tradeoffs between available parallelism for the OSA scheme, and the ability to exploit the *locality of reference*, as in the Scatter-Gather approach.

2.5 Overlapped-Overlapped Scattered Arrays

We are interested in exploiting the *locality of reference* in order to achieve higher execution speed in each individual processor. If this approach does not restrict the available concurrency too much, we should consider keeping the target rows in a fast, two-way interleaved SRAM in order to achieve twice the speed of the configurations described in Sections 2.2.2 and 2.4.3 in each processor. In this case, the execution of a particular row-wise operation is enabled by the completion of the tasks it depends upon, and by the availability of the target row in scattered form in the SRAM.

We initially assumed that a fixed number R of rows could be scattered in SRAM, which would require $R \times n$ words of storage. This situation corresponds to a very inefficient ($< 1\%$) usage of the fast memory due the sparse nature of the matrix. A severe restriction on R will cause in turn a severe restriction in the amount of concurrency available, as we shall demonstrate in the next section. In order to obtain a better utilization of the memory, and consequently obtain higher degree of parallelism, we propose a marching OSA structure, in which we try to fit in the fast memory as many rows as possible. We call this technique Overlapped-overlapped Scatter Array (O^2SA).

Figure 2-12 depicts the O^2SA representation of the small matrix used in our previous examples. Comparing this organization with the OSA structure shown in Figure 2-9, it is worth mention that the space required is substantially smaller; only the rows that are used as targets in update operations must be scattered (in this case rows 3, 5, and 8); and the structure changes are tightly *controlled* by the scheduling scheme: In the beginning of the execution, rows 3 and 5 are fitted in the array, and row 8 is left out because there is no space available. Even after the normalization of row 5 is executed, row 8 cannot fit in the structure. Only after the third row is discarded that it becomes possible to fit row 8. In other words, there is an *adaptive* utilization of the available memory space, depending on the structure of the rows and the scheduling heuristic. We will further discuss scheduling heuristics, and in particular the changes necessary in the scheduling for accommodating the O^2SA storage methodology in

$$\begin{array}{c|cccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
1 & X & & & & & & X & \\
2 & & X & & & X & & X & \\
3 & X & & X & & & & X & \\
4 & & & & X & X & & & \\
5 & & X & & & X & & X & \\
6 & & & & & & X & X & X \\
7 & & & & & & & X & X \\
8 & X & X & X & & X & X & X & X
\end{array}$$

Row	Offset
1	●
2	●
3	2
4	●
5	-1
6	●
7	●
8	0

when	1	2	3	4	5	6	7	8	9
initial	a_{52}	●	a_{31}	a_{55}	a_{33}	a_{57}	●	●	a_{37}
after $N5$	●	●	a_{31}	●	a_{33}	●	●	●	a_{37}
after $N3$	a_{81}	a_{82}	a_{83}	●	a_{85}	a_{86}	a_{87}	a_{88}	●

FIGURE 2-12: O^2SA representation of a matrix

Section 2.5.3.

2.5.1 O^2SA Performance

In order to evaluate empirically the requirements for the O^2SA, a valuable tool is to measure the processor utilization by restricting the number of scattered rows available on fast memory. Assuming R rows scattered at the same time in the fast memory, it is possible to use the scheduler to predict the processor utilization for the test matrices. A detailed description of scheduler used was given in Section 2.3.

Figure 2-13 depicts the processor utilization for the test matrix *dram* for 1 (or Scatter-Gather), 2, 4, 8, 16, 32, and 64 simultaneously scattered rows present in fast memory. According to Figure 2-13, at least a dozen active target rows, if not more, are necessary to achieve a reasonably high processor utilization. On the other hand, for practical purposes we would like to restrict the cache size to a maximum of two to three times n elements in order to be able to accommodate very large matrices with commercially available fast SRAMs. If these rows are simply scattered in separate memory positions, the utilization attained would be so poor that it would not justify the usage of a parallel processor.

In order to achieve our goal of keeping the destination row data in SRAM memory in order to achieve fast processing within each processor, we must radically change the way we perceive the usage of the SRAM space. Instead of a constrained set of R target rows, one should envision the expensive SRAM space as a cached O^2SA structure, which evolves with the execution of

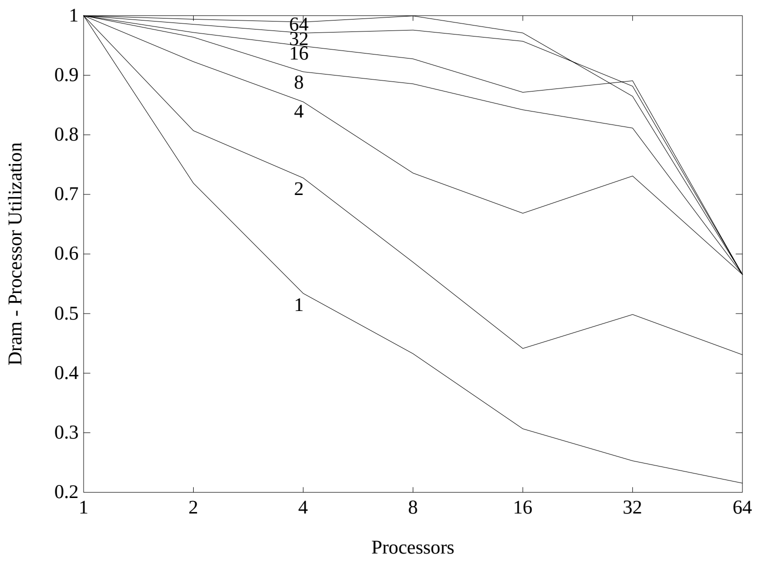


FIGURE 2-13: Processor utilization using up to R active targets on fast memory

the factorization, as shown in Figure 2-12. In the beginning of the factorization, dozens of small rows are scattered and overlapped, fitting in a small SRAM space of size $O(n)$, allowing a high degree of freedom for the parallel scheduling. Scattering into this small SRAM, in the other hand, insures the fast code execution within each processing element. After all the updates to a target row are finished, its elements are gathered back in a dense vector, and then other targets can be scattered in the SRAM. The factorization proceeds in this fashion, thus dynamically trading available SRAM space for concurrency.

Figure 2-14 depicts the estimated processor utilization for the LU decomposition of the test matrix *dram*, using the O^2SA technique for SRAM sizes of $2n$, $4n$ and $8n$. Considering the space occupied by the O^2SA structure, there is a substantial improvement relative to the utilizations depicted in Figure 2-13. This large improvement in the SRAM space requirements will enable us to design a parallel computer with specialized datapath and control in each processor geared towards the execution of the sparse matrix decomposition. This processor, briefly introduced in the next section, and described in detail in Chapter 4, profits from the O^2SA strategy by keeping the target rows in a fast, interleaved SRAM memory which is tightly coupled with the floating point unit.

2.5.2 O^2SA-Based Fast Sparse Matrix Decomposition Hardware

Figure 2-15 depicts a row update operation being executed in a special purpose processor, with dedicated datapath for O^2SA addressing. In essence, this organization is the same as that described for OSA addressing, as we use specialized units for address computation, special paths for column indices and we rely on memory interleaving for increasing the bandwidth.

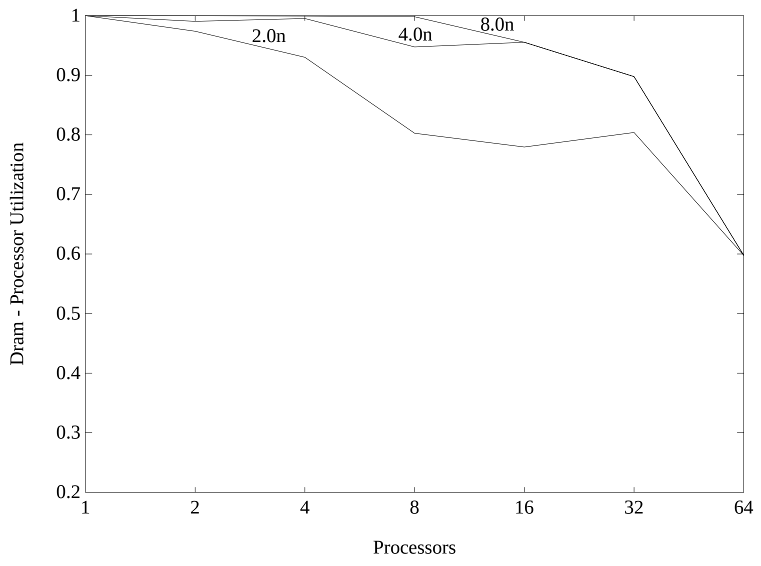


FIGURE 2-14: Processor utilization using the O^2SA structure to keep as many as possible active targets on fast memory

The combination of scheduling and storage proposed by this thesis opens the path for keeping the target rows in a smaller, faster SRAM memory. Also, as discussed in Section 2.2.1, two-way interleaving can be used very efficiently with target rows to double the processor speed. Two-way interleaving will also be required for the DRAM that holds the source data and the column indices, in order to match the speed requirements of the processor. Obviously, this interleaving exhibits even higher efficiency due to the sequential nature of the access to the source rows.

The system operates with two synchronized clock signals. One is fast and drives the floating point unit, the control unit and the SRAMs. The second, at half speed, controls the DRAMs, as these devices are slower. This constraint requires that the access to different DRAM memory banks be skewed by a half cycle.

The update operation depicted in Figure 2-15 proceeds in a pipelined fashion. The pair $\{q, a_{kq}\}$ is read from the dense source row vector in DRAM in the first half of a DRAM major cycle. In the second half cycle $\{r, a_{kr}\}$ are read, corresponding to the situation shown in the Figure 2-15. a_{kq} is multiplied by a_{ik}, while q is added to the offset *off* in order to access the even SRAM memory. The address previously generated off $+\ p$ is used to read the matching target data a_{ip} from the *odd* SRAM memory bank. The result of the multiplication that started two cycles before $(a_{ik} \times a_{kn})$ is subtracted from the data previously read (from the even SRAM memory bank) a_{in}. The ALU output register contains the results of the subtract operation initiated two cycles before $a'_{il} = a_{il} - a_{ik} \times a_{kl}$, which in turn are being written back to the *even* SRAM memory bank. In addition to all these operations, the read pointer to DRAM is being incremented and a counter is being decremented to check the size of the operation and

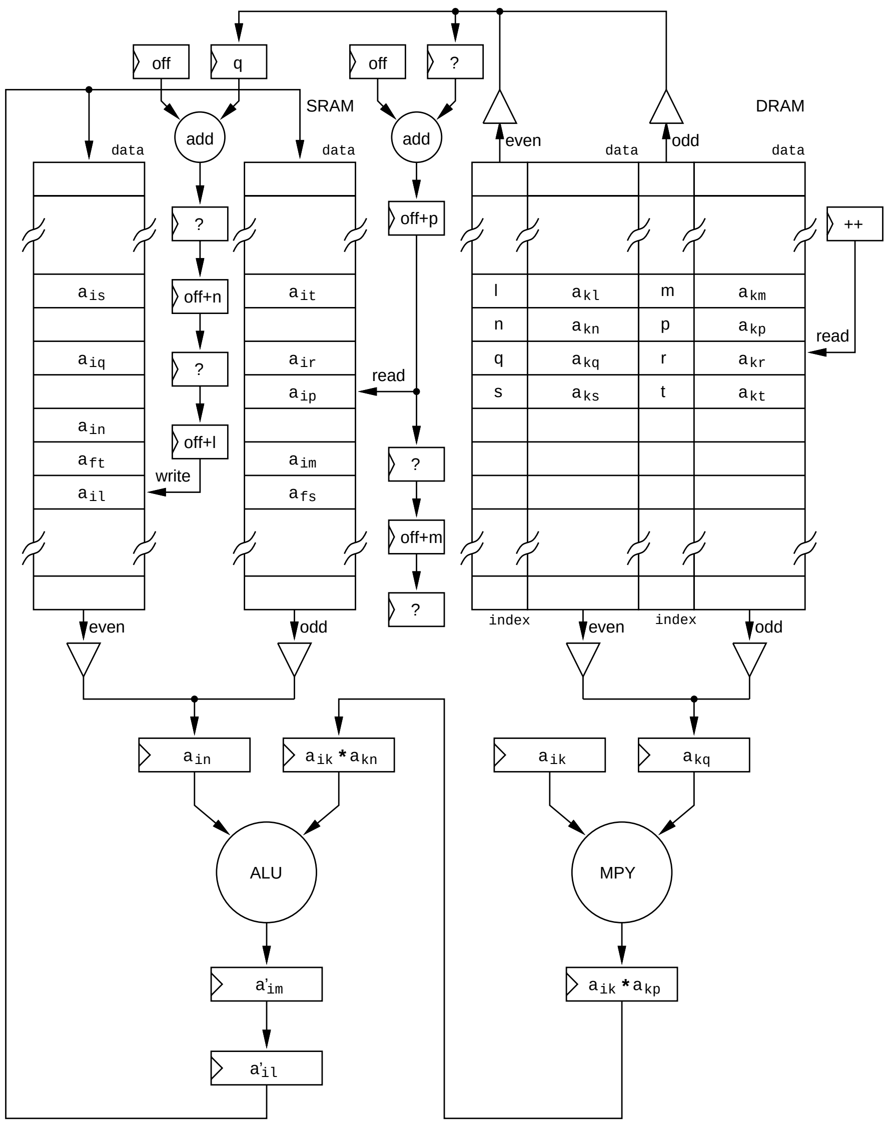


FIGURE 2-15: Dedicated datapath for fast O^2SA update

a conditional jump is taken to stay in the loop if the counter did not reach zero. Note that during each DRAM clock cycle twenty operations are executed — six integer operations, four floating point operations, eight memory accesses and two conditional jumps.

Table 2-9 compares the performance of the O^2SA-based sparse matrix hardware described in this section with the OSA-based sparse matrix hardware described in Section 2.4.3. The data was obtained by a detailed RTL simulation of the architectures shown in Figures 2-11 and 2-15. We also assumed that both systems are using the same memory technology: the DRAMs with a 40ns page-mode read/write cycle and 100ns random read/write cycle, and the SRAMs with a 20ns random read/write cycle. A more detailed description of the components and the overall processor architecture is provided in Chapter 4.

The first data set in the Table 2-9 corresponds to the OSA-based hardware and the second

Matrix	OSA $t_{CYCLE} = 40ns$			O^2SA $t_{CYCLE} = 20ns$		
	Cycles	PE Stall	Time(ms)	Cycles	PE Stall	Time(ms)
dram	1,523,775	351,124	61.0	1,286,943	137,132	25.7
feb	1,791,373	910,443	71.7	1,106,794	219,625	22.1
mesh	607,502	148,763	24.3	503,960	69,942	10.1
iir12	3,304,100	926,558	132.2	2,669,872	352,385	53.4
iir123	4,946,384	1,539,579	197.9	3,892,886	576,023	77.9
omega	519,517	261,493	20.8	340,756	82,224	6.8
mfr	1,300,575	528,177	52.0	960,518	214,249	19.2
Average	1,155,276	386,425	46.2	846,227	147,225	16.9

Table 2-9: Comparison of O^2SA-based and OSA-based sparse matrix hardware accelerators

data set corresponds to the O^2SA-based hardware. In each data set, the first column lists the total number of cycles necessary for the completion of the sparse LU decomposition. The second column lists the number of cycles in which the processor was stalled either due to a interleaved memory bank miss or because of a DRAM static column miss. The third column represents the expected time to complete the factorization, obtained by multiplying the total number of cycles by t_{CYCLE}. From the table, if it were possible to avoid the stalls, the cycle count for both schemes would be roughly the same. However, due to the difficulty in exploiting the *locality of reference* and the excessive number of DRAM static column misses, the overall number of clock cycles for the OSA-based scheme is on average 36% higher than the clock cycle count for O^2SA-based scheme. The biggest improvement however, derives from the ability to keep the *target rows* in the fast SRAM memory, effectively doubling the *gaxpy* throughput. When both effects are taken into account, the O^2SA-based hardware can deliver on average 2.7$\times$ the speed of the OSA-based hardware with a comparable component count and technological constraints.

The O^2SA improvements are not only limited for the hardware accelerators described in this section. The same techniques could also be applied to a block-oriented main memory (DRAM), and a small SRAM cache found in most computers today.

Having achieved the goal of exploiting the *locality of reference*, which allows a fast sparse matrix decomposition in each individual processor, the next section addresses the methodology used to schedule the sparse matrix factorization operations in a multiprocessor environment using the O^2SA storage mechanism.

2.5.3 Scheduling for O^2SA

Algorithm 2.7 uses a δ-greedy heuristic coupled with a simple scheme to support the concept of *active target rows*: one more dependency is added to the updates, as a row update operation can happen only if the corresponding target row has been scattered. The number of rows is not fixed *a priori:* the O^2SA strategy tries to overlap-scatter as many rows as possible in the small SRAM or cache space.

Algorithm 2.7 (δ-Greedy O^2SA Algorithm).

```
p = 1
while $p \leq P$ {
   SORT in $N_p$ all tasks $Ni \in X_p$ in decreasing order of $\delta_{Ni}$
   SORT all tasks $v_i \in X_p$ in decreasing order of $\delta_i$
   trytofit($p$)
   $p = p + 1$
   }
while $\bigcup X_p \neq \emptyset$ {
   $p = 1$
   while $p \leq P$ {
      visit the next ordered task to be scheduled $v_i \in X_p$
      if ($nd_i$ == 0) { /* optionally insert greedy algorithm */
         ...
         equivalent code in Algorithm 2.4,
         computes $t_{if}$, $t_{ol}$, $t_{of}$, $t_{il}$ and subtracting one from $nd_j$ for
         all tasks $v_j$ such that $\exists e_{ij}^{s,m}$
         .....
         if ($v_i$ == $Ni$) {           /* if $v_i$ is a row normalize task */
            schedule a gather for row $i$ from $O^2SA_p$
            remove($i$, $O^2SA_p$)
            trytofit($p$)
            }
         $last = \max(last, t_{ol})$
         $X_p = X_p - v_i$
         }
      $p = p + 1$
      }
   }
```

One safe way to decide the order in which the rows are going to be scattered in the O^2SA structure is by the using the top rows first, and descending towards the bottom rows. A better heuristic, and the only other way we were able to rigorously prove that the generated schedule is deadlock-free is by establishing a sequence of "candidate" rows i for scattering, ordered in decreasing values of $\delta_{\mathrm{N}i}$ for the tasks $\mathrm{N}i$. All other attempts to change this order caused the scheduler itself to stay in an endless loop. The proof that the proposed scheme can never cause deadlocks is analogous to the proof of Theorem 2.1, and is not shown here.

The execution starts in the same fashion as the δ-greedy scheme. After the values δ_i were

*Algorithm 2.8 (***trytofit***(p) procedure).*

```
$fail$= false
while !$fail$ {
   visit the next ordered task normalize row i, or $Ni \in N_p$
   if (fit($i$, $O^2SA_p$)) {
      schedule a scatter for row $i$ in $O^2SA_p$
      mark($i$,$O^2SA_p$)
      foreach $v_j \in X_p$ such that $v_j$ is an update $k \rightarrow i$
         $nd_j = nd_j - 1$ /* enable the update */
      }
   else $fail$= true
   }
```

computed for all tasks v_i, the tasks are distributed to P processors into separate bins X_p. The normalize tasks Ni are also copied into P bins N_p. Each task v_i has an associated timestamp t_i, representing the time the task will start, and the number of dependencies nd_i that must be satisfied before v_i is initiated.

Normalize tasks are ordered with respect to decreasing values of δ in each processor. Using this ordering, the procedure **trytofit(p)** will attempt to fit in the limited SRAM space in each processor as many rows as possible, in order to keep the maximum amount of concurrency available. Every time a row normalize is going to be scheduled, it is necessary to also schedule a gather from the O^2SA. The row is then removed from the structure and, according to the previously described order, a set of new rows is tested to fit in the unused spaces of the O^2SA structure. If any row fits in the available space, a scatter is scheduled, and the update tasks will be enabled, as highlighted by the comment /* enable the update */ in the **trytofit(p)** procedure shown in Algorithm 2.8.

The procedures **mark**(i, O^2SA_p), **fit**(i, O^2SA_p), and **remove**(i, O^2SA_p) handle a large array of single bits containing as many entries as the O^2SA structure, in which a bit set corresponds to an occupied scattered entry. These procedures correspond to setting, checking if possible to fit in, and resetting the bit positions corresponding to the non-zero entries of row i. The implementation of these procedures is straightforward. However, the scheduling speed will depend fundamentally on the implementation of the procedure **fit**(i, O^2SA_p).

A nice feature of Algorithm 2.7 is that it *adapts* the amount of concurrency available with the size and pattern of the rows. Typically, in the beginning of the execution, it is possible to scatter many short rows, allowing a lot of parallelism. As bigger rows are brought to the SRAM, the freedom of the scheduler algorithm decreases. In the worst case, a single row will be present in the SRAM, and the algorithm will behave as in the Scatter-Gather case. As previously discussed, the experimental results indicate that there is no substantial difference between the concurrency available using the OSA structure and O^2SA structure if the available SRAM (or cache) space is at least $2n$ words.

In the next section, we will discuss how we used the scheduler and its ability to predict the performance of a specific computer, given the appropriate cost functions, to perform some theoretical experiments and to fine tune the final version of architecture, described in detailed in Chapter 4. These results are validated by the comparison of the performance predicted by the scheduler and actual measurements made in a parallel processor and in a general purpose workstation.

2.6 Implementation Issues for Parallel Sparse Matrix Factorization on a General Purpose Multiprocessor

The task model described in Section 2.3.1 can represent with reasonable accuracy the underlying characteristics of the processor hardware and network interconnection, given the appropriate cost functions c_i and p_i. This approach has two advantages: first, a scheduler with insight into the hardware characteristics of the processor produces better code sequences; and second, it provides us feedback information about the performance that can be obtained using a specific configuration, which in turn helps in fine tuning the processor design.

The following discussion addresses some issues that are relevant for a real multiprocessor implementation, and illustrates the usage of the scheduler as a simulation tool. Specially relevant for us is the ability to predict the performance of a hypothetical machine and the effect of varying hardware parameters like the floating point unit latency, and the bus throughput. Many other issues can be easily addressed, as the scheduler software is able to *bind* the internal data structures with user supplied cost functions in a C-style syntax.

2.6.1 Influence of Hardware Parameters on Multiprocessor Performance

An issue that might interest a processor designer is the effect of the floating point pipeline depth on the performance of the system. Figure 2-16 depicts the processor utilization during the factorization of *dram* as a function of the number of processors for pipeline latencies of one, two, four, etc. clock cycles. The plot shows that a special purpose floating point unit for this application should have a large number of pipeline stages in order to operate with the highest possible throughput, as the performance degradation due to the pipeline latency is less than 5% in the worst case.

A simple explanation for this characteristic is that by having a deeper floating point pipeline the system is in fact exploiting in part the elemental or fine-grain concurrency available in the sparse matrix solution. Even though this approach does not provide the ultimate levels of concurrency, it substantially bridges the gap between the row-wise based scheduling and the elemental-level scheduling, exploited in the work of Huang and Karmakar [Huang79, Dhillon91].

The degree of parallelism can be measured by the depth of the pipeline times the number of processors times the utilization. For example, assuming fifty processors with a pipelined floating point unit eight levels deep, according to Figure 2-16, the utilization is around 75%. In other words, this would be equivalent to a level of concurrency of $50 \times 8 \times 0.75 = 300$.

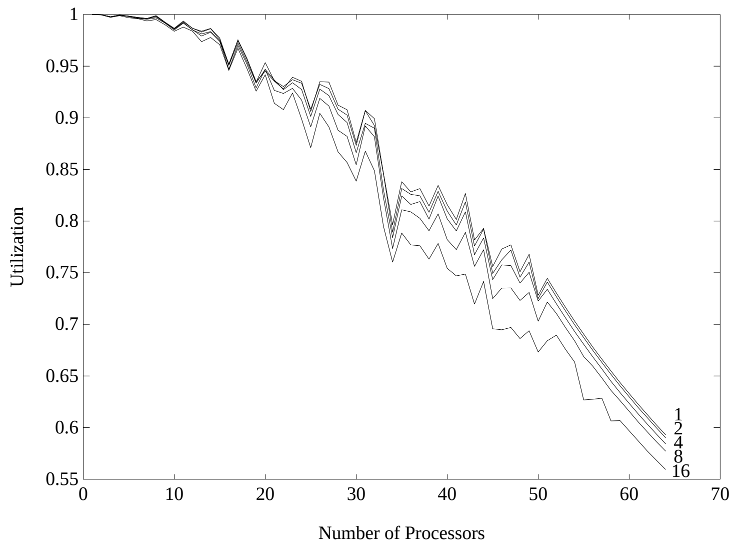


FIGURE 2-16: Effect of the pipeline latency on system performance

Since the design of a deeply-pipelined, extremely fast floating point unit is far beyond the scope of this thesis, we leave this issue as a topic for future research.

Another issue with more immediate repercussions is the impact of the network structure and bandwidth on the processor performance. We can model a high speed bus interconnection with the addition of bus broadcast tasks to the original task graph. These tasks are inserted between Ni and $i \rightarrow k$ and they must be executed in a special "processor" that represents the bus. This approach is consistent with the general task model, and therefore the scheduling schemes presented in Section 2.3.3 can be used without any modifications. Another advantage of this approach is that the bus transactions will also be ordered so that the total completion time is minimized. Figure 2-17 depicts the processor utilization during the factorization of *dram*, as a function of the bus bandwidth. The top line represents a theoretical network with no latency and infinite bandwidth. The middle line represents the utilization achieved if the bus has the same throughput as the floating point pipeline. In other words, the bus must be able to transmit a double precision floating point number in the same period that the floating point unit can compute one add and one multiply. Finally, the bottom line corresponds to the utilization achieved if the bus has half the bandwidth of the floating point pipeline. These results suggest that for a prototype implementation containing a small number of processors ($4 \leq P \leq 16$), it is not worth spending a lot of time designing a very high speed bus, as only a minimal advantage can be obtained by a running the bus any faster than half the processor speed. The bus interface subsystem described in detail in Chapter 4 was designed in view of these findings.

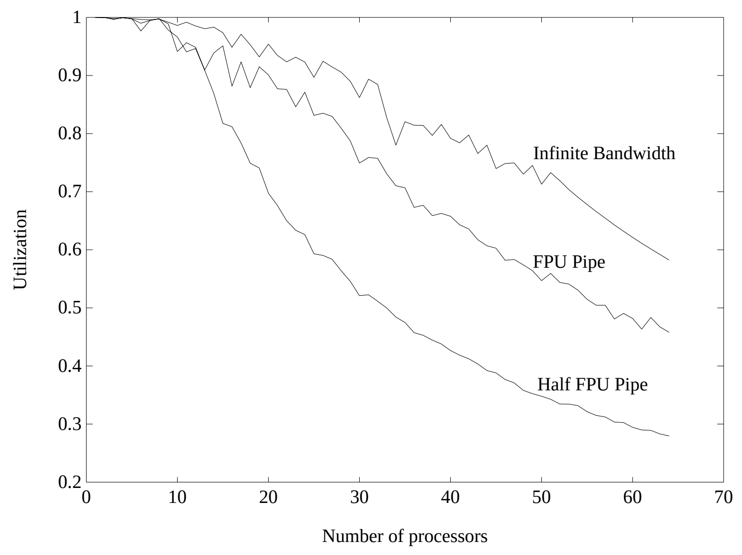


FIGURE 2-17: Effect of the bus bandwidth on system performance

Another important issue is the validation of the scheduler predictions, so that one can trust the results previously described. In the next section, the scheduler is validated using results from more detailed architectural simulations and from actual measurements.

2.6.2 Scheduler Validation

In order to validate the scheduler predictions, we have created an ensemble of tests designed to experimentally compare the performance predicted by the scheduler with actual measurements made in real machines, or with the results of a detailed RTL-level simulator for the architecture proposed in this thesis.

Table 2-10 compares the scheduler predicted values with the measured values for a RISC-based workstation, the IBM RS6000/540. The sparse matrix solver, a slightly modified version of Algorithm 2.6, was compiled, a list of the machine instructions was printed, and the appropriate cost functions for the row-wise update and the normalize tasks were hand-calculated. The cost functions are given in terms of clock cycles, which corresponds to 30ns for the IBM RS6000/540. The analysis of the inner loop of the sparse matrix code yielded the following clock cycle cost functions:

$$\begin{aligned} t_{UPDATE} &= 20 + 7 \times size_{SOURCE} \\ t_{NORMALIZE} &= 35 + 5 \times size_{SOURCE} \end{aligned} \quad (2.17)$$

Matrix	Predicted (ms)	Measured (ms)
mesh	88	90
dram	237	250
feb	133	170
iir12	474	670
iir123	676	950
omega	36	40
mfr	132	170

Table 2-10: Validation of the task model in an IBM RS6000/540 workstation

Table 2-10 indicates that for small matrices that fit totally in the processor cache, like *mesh*, *dram*, and *omega*, the values predicted by the scheduler were correct within 10%. Since the scheduler does not account for cache misses, the predicted values for large matrices were off by up to 40%.

Another important confirmation of the scheduler results comes from the PACE system, developed at AT&T by Prathima Agrawal and John Trotter[Agrawal92]. PACE is a distributed memory multiprocessor designed to speed up the LU decomposition of sparse matrices. The prototype machine consists of four Intel *i860* processors interconnected by a wide, high speed bus. Table 2-11 [Agrawal92] shows the factorization times for some of the test matrices previously described. The first and second columns indicate respectively the matrix and the number of processors used to factor it. The third and fourth columns show respectively the speedup predicted by the scheduler and the corresponding speedup measured in the PACE hardware. For *dram* and *iir12*, all results match within 5%. In the worst case, *omega*, the predicted performance was rather optimistic, as the speedup predictions were off by 15%. Column 5 shows the actual processor utilization, which in most cases is reasonably high. The actual decomposition time is in shown in the sixth column.

Overall, the results presented in Table 2-11 confirm that the processor utilization measured on PACE hardware is nearly in agreement with the simulated scheduling experimental results. More important, however, the results show that the scheduler is able to correctly generate code for the efficient sparse matrix decomposition on a multiprocessor system.

Finally, another validation of the scheduler results comes from a detailed simulation of the architecture proposed in this thesis. The first and second columns of Table 2-12 are respectively the matrix and the scheduler predicted performance, in clock cycles. The third column lists the total number of clock cycles obtained by a detailed RTL simulation of a single processing element. The fourth column list the number of cycles the processor was stalled, due to DRAM's static column misses and interleaving memory bank conflicts. Since the scheduler is not able to cope with these stalled cycles it is meaningful to compare the total number of valid cycles with the predicted values from the scheduler. It is also important to understand to which extension

Matrix	P	Speedup		Actual	Decomposition
		Predicted	Actual	Utilization	Time (s)
dram	1	1.00	1.00	1.00	0.81
	2	1.92	1.88	0.94	0.43
	3	2.65	2.61	0.87	0.31
	4	3.55	3.38	0.85	0.24
mfr	1	1.00	1.00	1.00	0.58
	2	1.85	1.71	0.86	0.34
	3	2.67	2.42	0.80	0.24
	4	3.37	2.90	0.73	0.20
iir12	1	1.00	1.00	1.00	2.03
	2	1.94	1.85	0.93	1.10
	3	2.70	2.67	0.89	0.76
	4	3.41	3.33	0.83	0.59

Table 2-11: Experiments on the PACE hardware

the stalled cycles degrade the processor performance.

According to Table 2-12, the number of valid cycles obtained by a detailed simulation is on average within 1% of the value predicted by the scheduler, even though on very sparse matrices like *feb*, and *omega*, the difference can be as big as 7%. On average, the number of cycles that the processor is stalled is 15%, tough in matrices that are very sparse like *omega* the stalled cycles can account for as much as 24% of the total number of cycles. The results from a detailed simulation are consistent with the proposed scheduler task model. In the future, a task model able to take into account the interleaving memory bank conflicts and the DRAM's static column misses could not only produce results closer to a more detailed simulation, but could also reduce the total number of clock cycles.

Matrix	Scheduler	RTL Simulation	
	(Predicted)	Total Cycles	PE Stalled
dram	1,131,755	1,286,943	137,132
feb	826,309	1,106,794	219,625
mesh	438,760	503,960	69,942
iir12	2,318,158	2,669,872	352,385
iir123	3,318,758	3,892,886	576,023
omega	249,027	340,756	82,224
mfr	741,427	960,518	214,249
Average	1,289,256	1,537,390	235,940

Table 2-12: Validation of the task model for the proposed architecture

Having discussed several storage and schedule methodologies, and validated some of the predicted performance results, we expect that the combined use of these software techniques, along with special purpose hardware for executing twenty operations per DRAM cycle will enable us to achieve in each processor execution speeds one order of magnitude faster than available general purpose processors driven by comparable clock speeds. The scheduler predictions and validations support the claims that another order of magnitude increase in performance can be obtained by the usage of multiple processors connected through a high speed bus.

In the next chapter, we intend to study techniques to use efficiently the computational resources for more generalized computations that are required for the evaluation of nonlinearities, as discussed in the Introduction. We intend to further discuss the proposed architecture in Chapter 4.

3

Parallel Circuit Simulation

The objective of this research is the study of software and hardware parallel execution techniques to speed up the simulation of electrical circuit. The main components that comprise the bulk of the circuit simulation computation are the assembly of a sparse set of network equations and the sparse matrix solution. While Chapter 2 describes several techniques for parallel sparse matrix decomposition, this chapter focuses on the aspects of parallel assembly of the network equations. The assembly process consists of the evaluation of all the circuit devices and the stamping of their contributions in the network equations.

Section 3.1 provides some background information about circuit simulation in general, along with an overview of the parallel circuit simulation process and the major issues needed to be addressed in order to achieve a high degree of multiprocessor utilization during the assembly of the network equations. Section 3.2 provides a detailed description of the data structures, equations and scheduling schemes used in a modified version of SIMLAB[Lumsdaine90] to implement the parallel device evaluation and contribution stamping. Finally, Section 3.3 presents predictions for the network equation assembly multiprocessor performance, along with a discussion on the merits of different scheduling schemes. The predictions are based on a relatively simple task graph cost function model similar to the one described in Section 2.3.

3.1 Major Issues in Parallel Circuit Simulation

3.1.1 Circuit Simulation Background

The equations that govern the dynamic behavior of electric circuits are the conservation laws, like Kirchoff's current (KCL) and voltage laws (KVL), the constitutive relations or current-voltage characteristics of the elements, and the physical correlations of rates of change of charge and flux with current and voltage. Let us consider, for example, the circuit in Figure 3-1.

Applying the conservation laws (KCL) for this circuit we obtain the following equations:

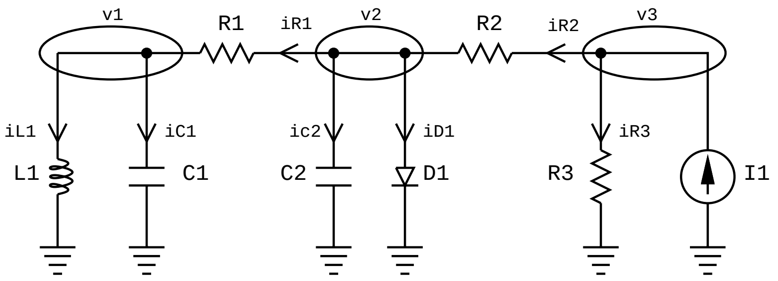


FIGURE 3-1: Example circuit

$$\begin{aligned} i_{R1} - i_{L1} - i_{C1} &= 0 \\ i_{R2} - i_{R1} - i_{C2} - i_{D1} &= 0 \\ i_{I1} - i_{R2} - i_{R3} &= 0 \end{aligned} \tag{3.1}$$

The following equations represent the constitutive relations, or the current-voltage characteristics of the elements:

$$\begin{aligned} i_{R1} &= \frac{v_2 - v_1}{R1} \\ i_{R2} &= \frac{v_3 - v_2}{R2} \\ i_{R3} &= \frac{v_3}{R1} \\ i_{D1} &= I_S \cdot (e^{\frac{v_2}{V_{th}}} - 1) \end{aligned} \tag{3.2}$$

The following equations represent the physical correlations of rates of change of charge and flux with currents and voltages:

$$\begin{aligned} i_{C1} = C1 \cdot \frac{dv_1}{dt} \\ i_{C2} = C2 \cdot \frac{dv_2}{dt} \\ v_{L1} = L1 \cdot \frac{di_{L1}}{dt} \end{aligned} \tag{3.3}$$

Relationships 3.1, 3.2, and 3.3 can be generalized for any network containing a mix of linear and non-linear lumped elements. These equations are represented as a sparse set of non-linear, algebraic-differential equations [Vlach83]:

$$F\left(\frac{dx(t)}{dt}, x(t), t\right) = 0 \tag{3.4}$$

where $x(t)$ is a vector containing voltages, currents, charges and fluxes, $\frac{dx(t)}{dt}$ represents the rate of change of these variables in respect with time, and $F()$ represents the set of non-linear equations linking $x(t)$, $\frac{dx(t)}{dt}$, and t associated with the circuit. Finding an appropriate value of $x(0)$ that is consistent with Equation 3.4 is known as the *DC solution* of the circuit. Given the DC solution, the *transient analysis* computes numerically the values of $x(t)$ for some $t > 0$.

The particular implementation of the circuit simulator used in this thesis is a modified version of SIMLAB [Lumsdaine90]. SIMLAB restricts to a small extent the flexibility of the simulator by allowing only charges and currents to participate in the formulation of $F()$. Since flux and voltage equations are not modeled, floating voltage sources cannot be represented in the model. In spite of its shortcoming, SIMLAB can simulate most devices present in an integrated circuit. Furthermore, the extension for a more generalized simulator is relatively straightforward.

During the transient simulation in SIMLAB, there is one non-linear differential-algebraic equation per circuit node that is not directly connected to a voltage source. Each equation relates the rate of change of the node charge with the current balance.

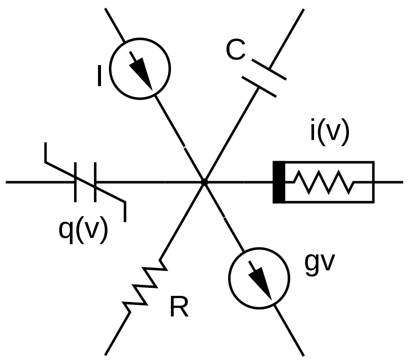


FIGURE 3-2: Examples of elements connected to a node

Let us consider, for example, the rate of change of charges in capacitors connected to all nodes in the circuit and the balance of currents entering and leaving these nodes. Figure 3-2 depicts several examples of devices connected to a particular circuit node. The following set of equations represent the circuit behavior of a generic SIMLAB network:

$$\frac{dq(v(t))}{dt} + C_0\frac{dv(t)}{dt} + i(v(t)) + G_0 v(t) = 0 \tag{3.5}$$

In Equation 3.5, $v(t)$ is the vector of node voltages, $q(v(t))$ represents the contribution of the non-linear capacitive elements, and $i(v(t))$ represent the contribution of the non-linear resistive elements. C_0 and G_0 represent respectively the linear capacitive and resistive elements. Assuming a known voltage vector $v(t-h)$, and a generic timestep h, this version of SIMLAB uses the trapezoidal integration method to approximate the time derivatives in Equation 3.5

by the average value of the current values at time t and $t-h$, which yields the following sparse set of non-linear algebraic equations:

$$\left[\frac{Q(v(t)) - Q(v(t-h))}{h}\right] + \left[\frac{I(v(t)) + I(v(t-h))}{2}\right] = 0 \tag{3.6}$$

where $Q(v)$ is the vector of charges, including the contributions of both linear and non-linear capacitive elements, while $I(v)$ is the vector of currents, including the contributions of both linear and non-linear conductances. Equation 3.6 can be solved using the multidimensional Newton-Raphson iteration scheme, in which the value of the k-th iterate $v^k(t)$ is given by solving the following sparse set of *linear* equations:

$$\begin{aligned}\frac{\partial}{\partial v^k(t)}\left[\alpha[Q(v^k(t)) - Q(v(t-h))] + [I(v^k(t)) + I(v(t-h))]\right]\left[v^k(t) - v^{k-1}(t)\right] = \\ -\alpha[Q(v^k(t)) - Q(v(t-h))] - [I(v^k(t)) + I(v(t-h))]\end{aligned} \tag{3.7}$$

where $\alpha = \frac{2}{h}$. This equation can be rewritten in a compact way as $J_F(v^k)\Delta v^k = -f(v^k)$. The Jacobian $J_F(v^k)$ consists of the partial derivatives of the components of $F()$ with respect to the components of $v(t)$, or $\left[\alpha\frac{\partial Q(v^k(t))}{\partial v^k(t)} + \frac{\partial I(v^k(t))}{\partial v^k(t)}\right]$. The right hand side $-f(v^k)$ consists of the contributions of current and charge to the error in the approximation of the iterate $v^k(t)$. The iterative scheme starts with some initial guess $v^0(t)$, which can be obtained by interpolation of the previous timestep values, and is repeated until $||\Delta v^k|| \leq \epsilon$ and $||f(v^k)|| \leq \delta$, given some arbitrary error tolerance ϵ and δ. Once the convergence is reached and $v(t)$ is obtained, a new timestep h' is picked, and the whole process is repeated to obtain $v(t+h')$.

Assembling these equations involves determining for all devices in the circuit the currents, charges, and their derivatives in respect with the voltage vector, a process often called *model evaluation*, and then adding their contributions to the Jacobian matrix $J_F(v^k)$ and the right-hand-side vector $-f(v^k)$. The processes of adding these contributions are called respectively *Jacobian* and *right-hand-side stamping*.

Algorithm 3.1 describes in detail one step of the transient analysis algorithm used in SIMLAB. The algorithm starts by choosing a value for the timestep h based on an estimate of the local truncation error. Also, an initial voltage vector v_g for the N nodes that are not connected to independent voltage sources is guessed by interpolating $v(t)$ from a small set of previous values of v. The vector $u(t)$ has N_{indep} entries corresponding to nodes with independent voltage sources, and it is updated to reflect their status at time t. The initial guess $v^0(t)$ is assembled by the concatenation of $v_g(t)$ and $u(t)$ vectors. The contributions of $-\alpha Q(v(t-h)) + I(v(t-h))$ in Equation 3.7 are combined in a single vector rhs^{old}. The algorithm then proceeds with the Newton-Raphson iteration.

During the Newton-Raphson iteration process, it is possible to substantially reduce the computation costs by evaluating the Jacobian entries only once in a cycle that consists of a

Algorithm 3.1 (Transient Analysis — Computation of one timestep).

```
$converged$= $v\_converged$= **false**
$h$= **pick_timestep()**                                    /* Choose timestep */
$t$= $t + h$
$\alpha = \frac{2}{h}$
$v_g(t)$= **guess_voltage**($t$)                            /* Compute initial guess */
$u(t)$= **compute_independent_Vsources**($t$)
$k = 0$
$v^k(t) = v_g(t) \bigcup u(t)$
$rhs^{old} = c - \alpha q$              /* Initialize $rhs^{old}$ with $-\alpha Q(v(t-h)) + I(v(t-h))$ */
while ($k < MaxIter$) and (!$converged$) {                  /* Newton-Raphson iteration */
    $do\_LU$= false
    if (($k$ % $NJacob$) == 0 ) {           /* Compute Jacobian once per $NJacob$ cycle */
        $C = \frac{\partial Q(v^k(t))}{\partial v^k(t)}$          /* Evaluation of derivatives using models */
        $G = \frac{\partial I(v^k(t))}{\partial v^k(t)}$
        $J_F(v^k(t)) = \alpha C + G$                          /* Jacobian stamping */
        $do\_LU$= true
        }
    $q = Q(v^k(t))$                                         /* Evaluation of device models */
    $c = I(v^k(t))$
    $rhs = c + \alpha q + rhs^{old}$                          /* $rhs$ stamping */
    if (!$v\_converged$) or (!check_Iconvergence($rhs$)) {
        $\Delta v^k(t)$= sparse_solve($J_F(v^k(t)), -rhs, do\_LU$) /* Solve linear equations */
        $v\_converged$= check_Vconvergence($\Delta v^k(t)$)
        if (!$v\_converged$) {                              /* If has not converged yet */
            $v^{k+1}(t) = v^k(t) + \Delta v^k(t)$
            $k = k + 1$
            }
        }
        else {
            $converged$= true
            }
    }
```

small number *NJacob* of iterations. This change does not change substantially the total number of iterations necessary to achieve convergence, yet substantially reduce the computational costs. Since the Jacobian entries do not change during the other iterations of the *NJacob* cycle, it is only necessary to perform a matrix LU decomposition when the Jacobian entries are updated, as indicated in Algorithm 3.1 by the *do_LU* boolean variable.

Besides the solution of the sparse set of equations, discussed in detail in Chapter 2, the bulk of the algorithm execution time is spent computing matrices C and G, respectively the capacitive and conductive derivatives that are added to form $J_F(v^k(t))$, and the vectors c and q, respectively the current and charge parts of rhs. The matrices C and G are not actually computed separately and merged during the stamping phase generating the Jacobian matrix $J_F(v^k(t))$. This approach would require a large amount of memory to store three very large matrices. Instead, the contributions are automatically merged and saved directly into the Jacobian matrix, as shown in detail for some devices in Section 3.2. However, vectors c and q must actually be separately generated before the right-hand-side vector is assembled, and must be kept separate until the next timestep, as their contributions are needed to generate the old right-hand-side vector $rhs^{old} = c - \alpha q$. This constraint, however, does not demand a large amount of memory.

3.1.2 Overview of the Parallel Circuit Simulation Process

Figure 3-3 depicts some of the data structures involved in the parallel execution of the model evaluation and the stamping process. Each processor contains a set of Jacobian matrix $J_F(v^k(t))$ rows that are selected by the sparse matrix partioning and scheduling algorithm, and the corresponding scattered set of entries in vectors $\Delta v^k(t)$ and $rhs(v^k(t))$. We assume that each processor contains a copy of the entire voltage vector $v^k(t)$. Each processor also contains a list of devices that will be evaluated locally. Each device instance has associated with it a model, the instance parameters, and an ordered list of addresses that correspond to the terminal voltages, and to the contributions for the Jacobian and the right-hand-side vectors c and q.

For example, processor #1 in Figure 3-3 contains in its device list a diode. The processor can examine the values of $v_a^k(t)$ and $v_c^k(t)$ in the local voltage vector $v^k(t)$ to evaluate the current c and charge q contributions for the anode a and the cathode c, as well as the four contributions to the Jacobian J_{aa}, J_{ac}, J_{ca}, and J_{cc}. In Figure 3-3, processor #P contains the data associated with the node 1, where the anode is connected to, while processor #1 contains the data associated with the node 2, where the cathode is connected to. Following the steps in Algorithm 3.1, after the models have been evaluated and their contributions to Jacobian and the right-hand-side vector have been updated in the different processors, the parallel sparse matrix solution algorithm described in Chapter 2 is executed, and the local values of $\Delta v^k(t)$ are obtained. The solution vector, augmented by extra zeros representing the N_{indep} nodes connected to independent voltage sources is then added to the present voltage values $v^k(t)$ in order to obtain the next iterate $v^{k+1}(t)$. The convergence is checked, and the next Newton-Raphson iteration is restarted.

The model evaluation is a parallel problem that can easily achieve near perfect speedup in most multiprocessor systems. The speedup is bounded only by the granularity of the tasks, which is usually very fine, as the number of devices M to be evaluated is usually much larger

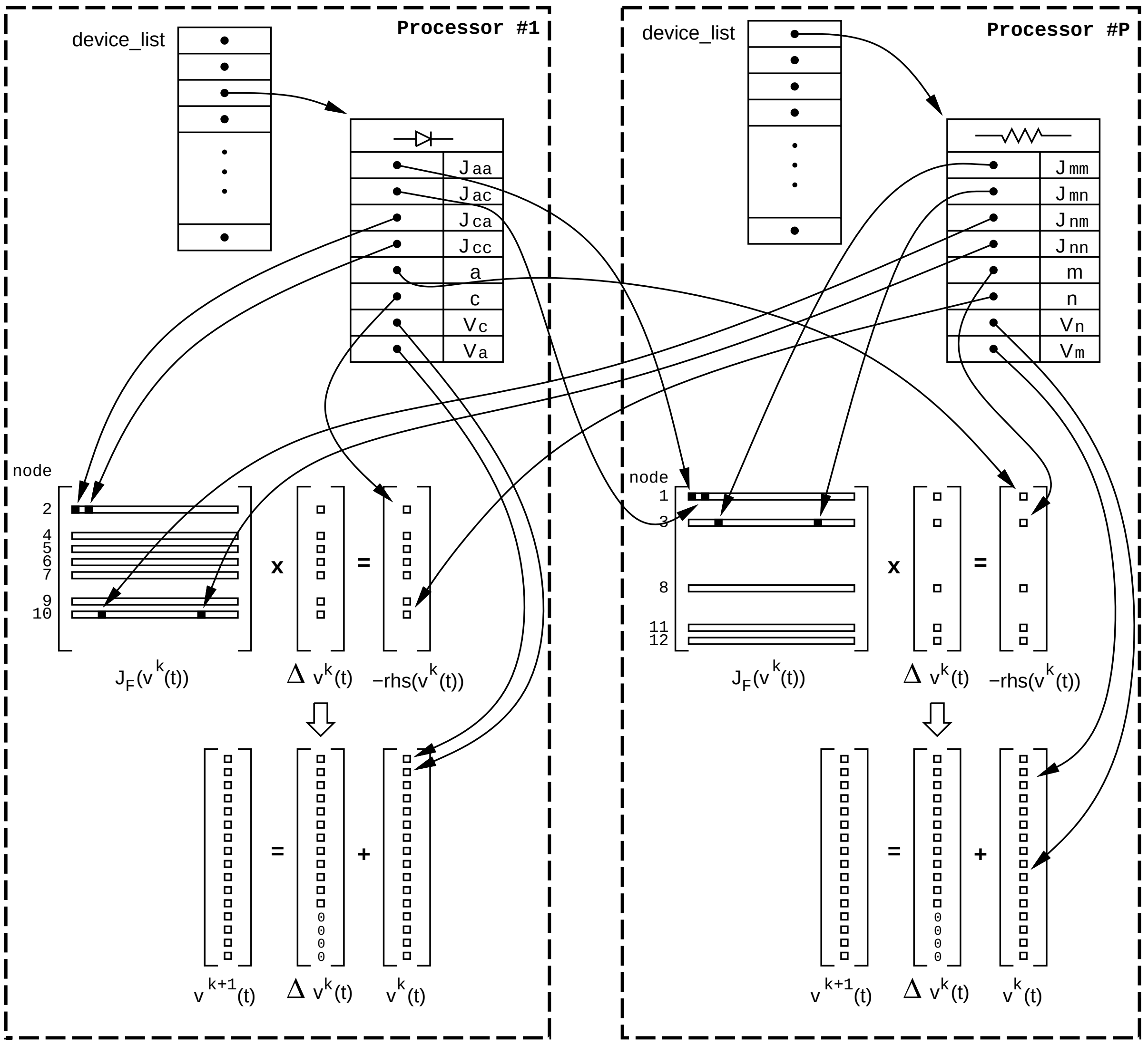


FIGURE 3-3: Data structures used in a multiprocessor for transient analysis of an electrical circuit

than the number of processors P. However, adding the different device contributions to the Jacobian $J_F(v^k(t))$ and to the right-hand-side vectors c and qs requires either the transmission of the contributions across the network, or the duplication of the device evaluation effort. For example, considering the diode in Figure 3-3, its contributions J_{aa} and J_{ac} for the Jacobian, as well as the anode current and charge contributions can either be transmitted from processor #1 to processor #P, or the diode evaluation can be done independently in processor #P, and the above mentioned contributions could be locally added.

Assuming that no evaluation task is replicated in different processors, implying that some data must be transmitted over the network to other processors, and the interconnection network topology consists of a simple bus, it is possible to envision two different situations, previously described in the work of Sadayappan and Visvanathan [Sadayappan88] for a shared memory multiprocessor. This simplified model permits a quick evaluation of the impact of the network bandwidth on multiprocessor performance. In order to extend Sadayappan's results for a distributed memory multiprocessor with message passing, let us consider a parallel loop executed in P processors, which consists of the evaluation of a very large number of compute intensive tasks corresponding to the model evaluation, each with a probability λ of being followed by a transmission of its results, so that if necessary, the contributions are added in the appropriate processor to the correct entry of the right-hand-side vector or the Jacobian matrix. Empirically, the value of $\lambda \approx 0.5$ is a reasonable assumption for large MOS circuits simulated in a large number of processors, a claim further discussed in Section 3.3. Figure 3-4 shows this loop: after a given task is finished, its corresponding output data might be transmitted in the bus. Assuming that the average time required for the device evaluation is t_{eval}, and the time for each transmission is t_{trans}, Figure 3-4 depicts the two possible scenarios. If $t_{eval} \geq \lambda \times P \times t_{trans}$, the speed-up obtained is P, which corresponds to an utilization of 100% of the computational resources. This situation is depicted in Figure 3-4(a). The empty time slots in the bus represent bus *slack*. On the other hand, if $t_{eval} < \lambda \times P \times t_{trans}$, the speed-up obtained is $\frac{t_{eval}}{\lambda \times t_{trans}}$, regardless of the number of processors available. We shall refer to this ratio as *maximum speedup due to stamping*, or s_{max}. The utilization in this case drops to $\frac{t_{eval}}{\lambda \times P \times t_{trans}}$. This situation is depicted in Figure 3-4(b). The empty time slots in each processor represent an *idle* processor. A clever scheduling algorithm can predict in advance the latter situation, and duplicate some of the tasks in order to alleviate the network traffic. The ideal operating situation is when all processors are all used most of the time, and the bus is always occupied, thus minimizing the total completion time for equation assembly.

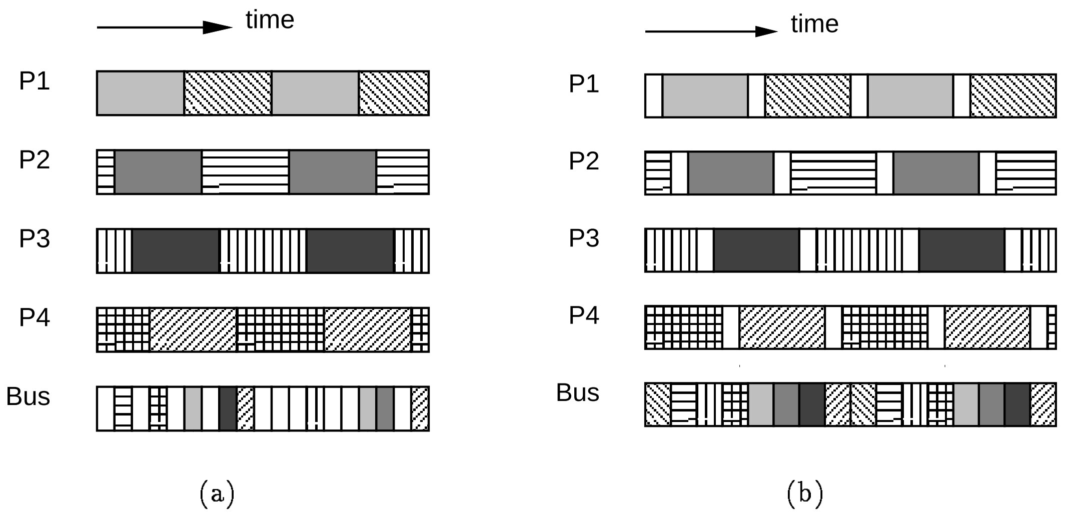


FIGURE 3-4: Bus *slack* (a) versus processor *idle* (b)

It is important to note that for certain devices, like linear elements and simple voltage sources, the evaluation costs t_{eval} are actually smaller than the t_{trans}. Let us consider, for example, the resistor in Figure 3-3. Assuming that $\frac{1}{R}$ has been precomputed, there is no cost for the evaluation of the contributions for the Jacobian J_F, and the cost for the contributions to the right hand side is one floating point multiplication, which is smaller than t_{trans} in almost all multiprocessor systems currently available. Since this is the case with most linear devices and independent voltage sources, the first version of the scheduler *replicates* the work for these elements if their contributions reside on different processors. On the other hand, we expect t_{eval} for non-linear devices like transistors and diodes to be much larger than t_{trans}, at least considering the Numerical Engine proposed in Chapter 4, so the first version of the scheduler computes only once the contributions of non-linear devices, and then transmit the results over the network for stamping. The next section discusses in detail the issues involved in parallel model evaluation and the issues involved in the final assembly of the equations by stamping the device contributions.

3.2 Parallel Model Evaluation

3.2.1 Linear Elements and Voltage Sources

Assuming the k-th iterate of the voltage vector $v^k(t)$ is present in all processors, it is rather straightforward to compute all contributions for the linear elements. Since most linear element computations are very simple, the evaluation and stamping are done in the same task, and no data is transmitted over the network for finishing the assembling the equations. Figure 3-5 depicts for each processor the data structures associated with the evaluation of linear elements and voltage sources, corresponding loosely to part of the device_list structure in Figure 3-3.

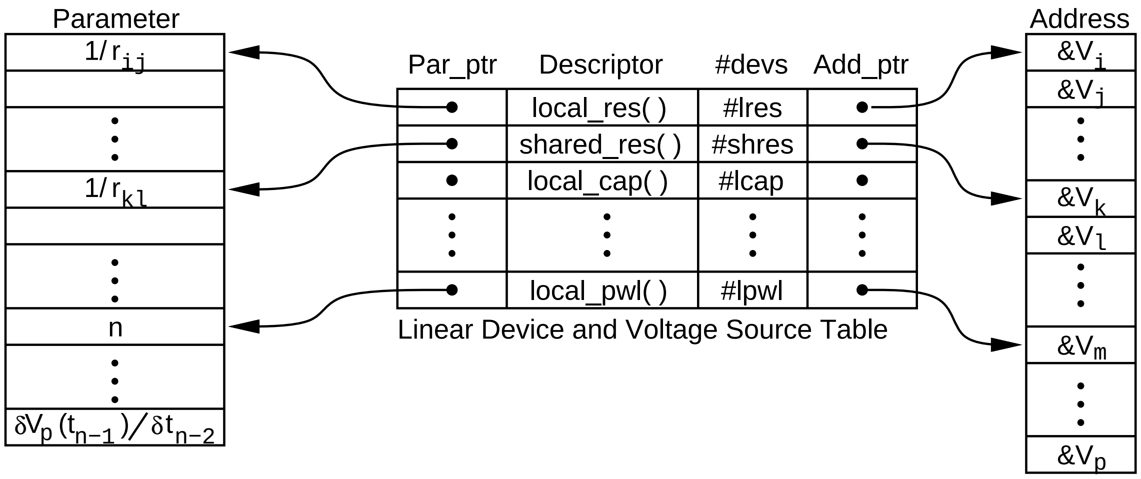


FIGURE 3-5: Local linear device and voltage source data structures

Once the partition algorithm for sparse matrices has decided in which processor a particular row (representing a circuit node) is stored, it is relatively straightforward to precompile a list of linear device evaluation tasks for each processor. In order to simplify the precompilation, all independent voltage sources are evaluated locally in each processor. Each type of device has one or more entries in the Linear Device and Voltage Source Table. Each entry in this table contains a descriptor field containing the address of the device evaluation procedure, the number of device instances, a pointer to a list of double precision floating point numbers representing instance parameters, and a pointer to list of addresses associated with the instances. For example, capacitors are either connected to two circuit nodes stored in the same processor, or connected to nodes stored in different processors. Therefore, there are two entries in the device table for capacitors. In each processor, for all capacitor instances whose nodes are stored locally, the local_cap() procedure evaluates and stamps their contributions for the Jacobian and the right-hand-side. For instances of capacitors whose nodes are stored in different processors, the shared_cap() procedure in each processor evaluates and stamps the local contributions. In the following, the evaluation and stamping of some of linear devices and independent voltage sources are described in detail.

Linear Capacitors

Figure 3-6 illustrates the data structures associated with each capacitor instance, for both local (a) and distributed (b) evaluation and stamping.

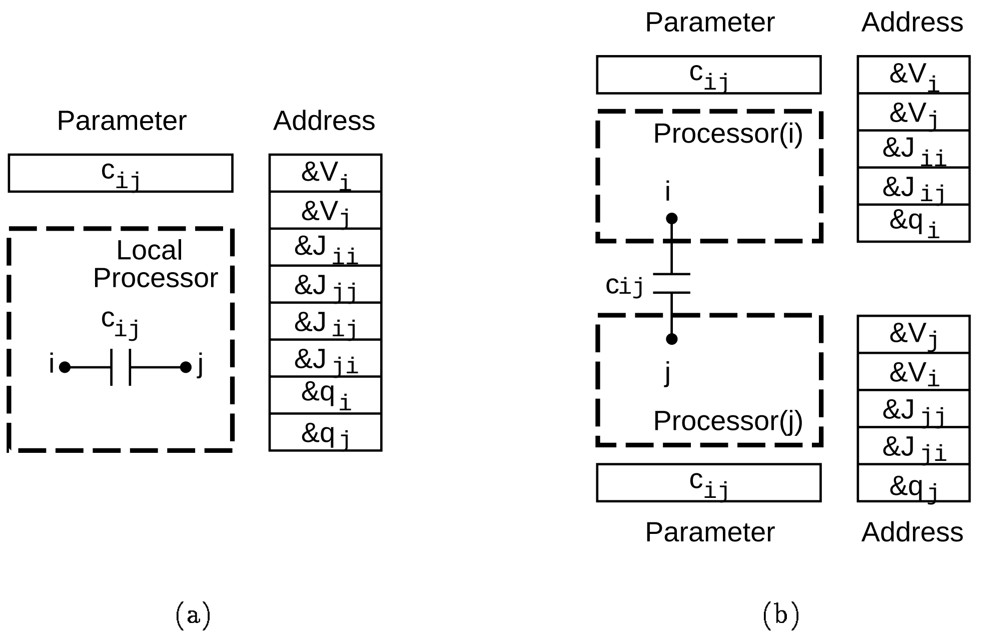


FIGURE 3-6: Local (a) and distributed (b) capacitor evaluation and stamp data structures

The constant $\alpha = \frac{2}{h}$ is described in Equation 3.7. For each capacitor instance terminal, Equations 3.8 and 3.9 describe the evaluation and the stamping in the Jacobian matrix and in the q vector, which represents the part associated with charges in the right-hand-side vector.

$$\begin{aligned} J_{ii} &= J_{ii} + \alpha c_{ij} \\ J_{ij} &= J_{ij} - \alpha c_{ij} \\ q_i &= q_i - c_{ij}(V_i - V_j) \end{aligned} \tag{3.8}$$

and

$$\begin{aligned} J_{jj} &= J_{jj} + \alpha c_{ij} \\ J_{ji} &= J_{ji} - \alpha c_{ij} \\ q_j &= q_j - c_{ij}(V_j - V_i) \end{aligned} \tag{3.9}$$

It is important to note that the form of Equations 3.8 and 3.9 is the same to simplify the implementation of the shared_cap() procedure. In order to accommodate this situation, the order of V_i and V_j addresses for the device instance in different processors is swapped, as shown in Figure 3-6(b).

Independent Current Sources

Figure 3-7 illustrates the data structures associated with each independent current source instance, for both local (a) and distributed (b) evaluation and stamping.

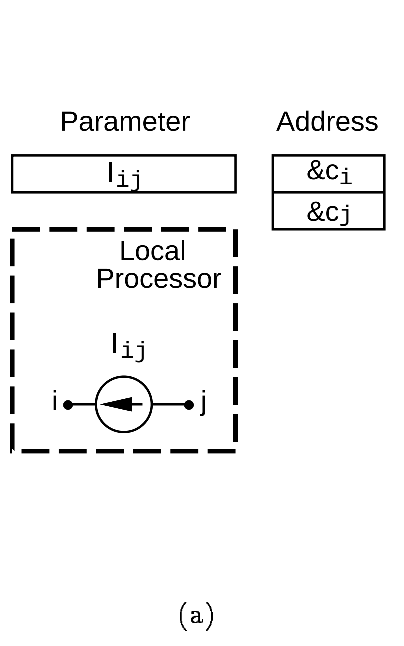
Equations 3.10 and 3.11 describe for each independent current source instance node the evaluation and the stamping in the right-hand-side vector. No Jacobian entries are affected. In order to simplify the implementation of shared_isrc(), the current parameters for the same instance in different processors have opposite signs, as shown in Figure 3-7(b).

$$c_i = c_i - I_{ij} \tag{3.10}$$

$$c_j = c_j - (\,-I_{ij}) \tag{3.11}$$

Independent Voltage Sources

Figure 3-8 illustrates the data structures associated with each independent voltage source instance. As the evaluation and stamping for these elements is trivial, all instances are local in all processors in order to simplify the precompilation software. The evaluation and stamping of the independent voltage source instances consists only of resetting the appropriate voltage V_i in the i-th position of the vector $u(t)$, during the execution of **compute_independent_Vsources(t)** procedure in Algorithm 3.1.



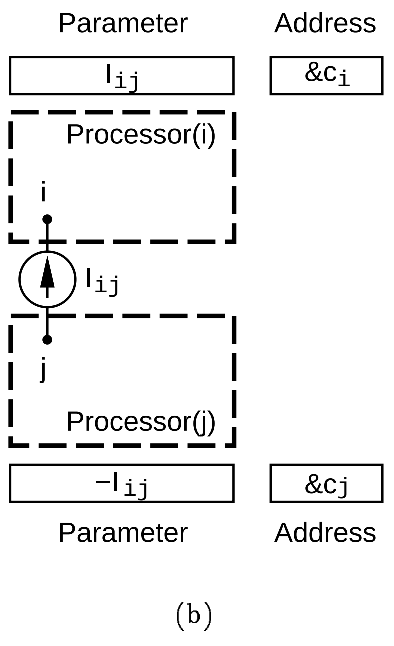


FIGURE 3-7: Local (a) and distributed (b) independent current source evaluation and stamp data structures

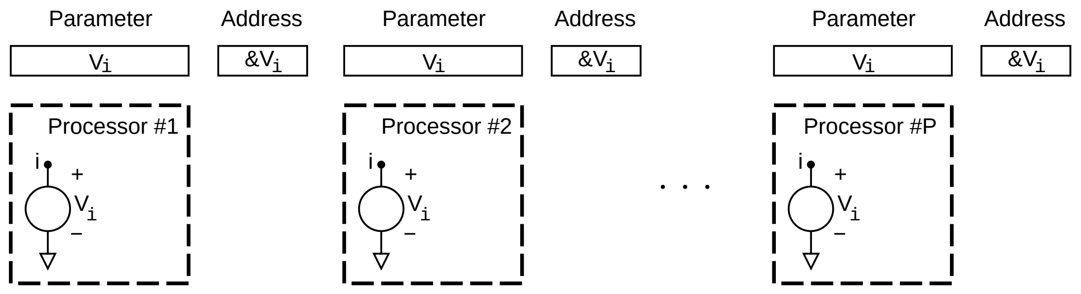


FIGURE 3-8: Distributed independent voltage source evaluation and stamp data structures

Piecewise Linear Independent Voltage Sources

Figure 3-9 illustrates the data structures associated with each independent piecewise linear voltage source instance. Like independent voltage sources, the evaluation and stamping for these elements is trivial, and all instances are local in all processors in order to simplify the precompilation software. Algorithm 3.2 describes how to obtain $V_i(t)$ given t during the execution of **compute_independent_Vsources(t)** in Algorithm 3.1. As shown in Figure 3-9, the set of parameters associated with each instance of a voltage source consists of n, the number of linear pieces that compose the waveform, its *delay* ΔT, its *period* T, and three vectors of size n whose components define precisely each linear piece. The three vectors represent respectively the initial time, the initial voltage, and the slope of each linear piece. The slope vector is added to accelerate the execution of the procedure, as floating point divisions are substantially more expensive than multiplications. The first time element $t_0 = 0$ is implicitly stored, so the first vector is actually skewed by one element in respect to the other vectors, and finishes with $t_n = T$. The total size of the data structure for each voltage source is $3n + 3$.

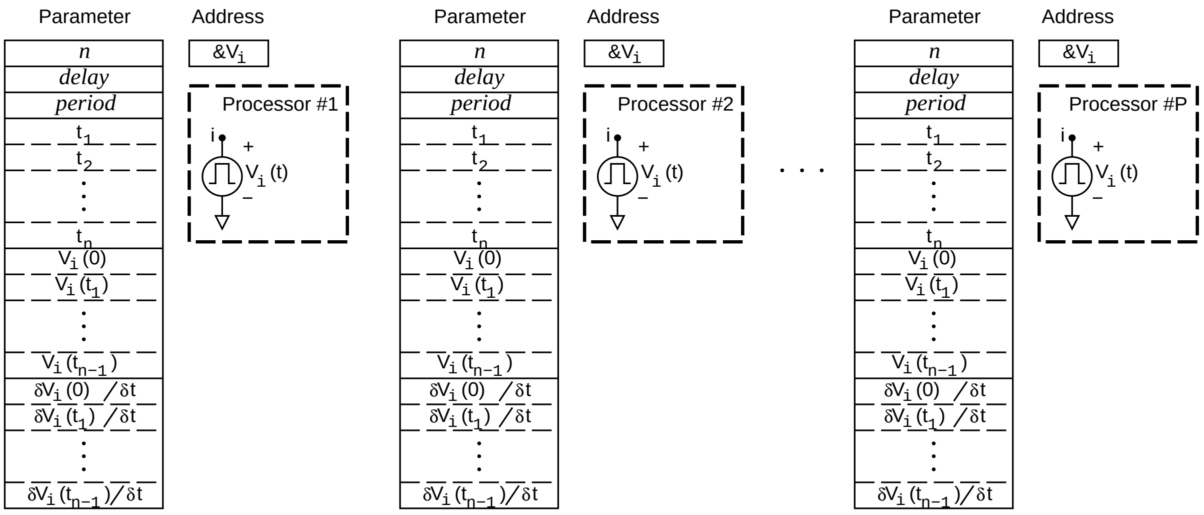


FIGURE 3-9: Data structures for the evaluation and stamp of the independent piecewise linear voltage sources

For each piecewise linear independent voltage source, Algorithm 3.2 computes the local time relative to t_0, searches for the appropriate linear segment j that corresponds to the local time, and obtains the voltage $V_i(t)$ by multipling the distance between t_{local} and t_j by the slope $\frac{\partial V_i(t_j)}{\partial t}$ and adding it to the initial voltage $V_i(t_j)$.

3.2.2 Parallel Evaluation of Non-Linear Elements

The parallel evaluation and stamping of non-linear elements is a substantially harder problem than its equivalent for linear devices. The time necessary for the evaluation of the contributions for a complex device like a diode or a transistor is much larger than the time required to transmit them in the Numerical Engine network described in Chapter 4. Therefore, the first version of the scheduling software assumes that the contributions for non-linear devices will be computed in only one processor and transmitted over the network for stamping. In the following, we present in detail the mechanics of precompiling a list of operations for the parallel device evaluation and contribution stamping using data transmission. The mechanics for task replication scheduling are trivial and therefore are not discussed. A short discussion on the predicted performance for different scheduling methods is presented in Section 3.3.

During the precompilation phase, each device instance is visited, and the least loaded processor that contains nodes connected to the instance under consideration is assigned for the evaluation of the contributions to the Jacobian and the right-hand-side entries. If any instance terminals are connected to nodes that reside in different processors, the appropriate contribu-

Algorithm 3.2 (Computing $v(t)$ for an independent piecewise linear voltage source).

```
local_pwl(Par_ptr, #lpwl, Add_ptr, t)

ptr= pbase= Par_ptr
abase= Adr_ptr
for $k = 1$ to #lpwl do {                                        /* Foreach pwl source */
    $n = *ptr++$                                          /* Number of linear segments */
    $\Delta T = *ptr++$                                                      /* Delay */
    $T = *ptr++$                                                            /* Period */
    $t_{local} = t - \Delta T - \lfloor \frac{t-\Delta T}{T} \rfloor T$                                            /* Local time */
    $j = 0$
    while $t_{local} > (t_j = *ptr++)$ {    /* Search for corresponding time segment */
        $j = j + 1$
        }
    $V_i(t_j) = pbase[n + j + 3]$                         /* Corresponding initial voltage */
    $\frac{\partial V_i(t_j)}{\partial t} = pbase[2n + j + 3]$                                /* Corresponding slope */
    $V_i(t) = V_i(t_j) + \frac{\partial V_i(t_j)}{\partial t} \times (t_{local} - t_j)$
    **abase = $V_i(t)$                                                        /* Stamp */
    abase= abase + 1                         /* abase points to next pwl source */
    pbase= pbase + $3n + 3$                        /* pbase points to next pwl source */
    }
```

tions will be transmitted over the network and stamped remotely, as shown in Figure 3-3.

The simple partitioning and scheduling scheme described above is not optimal, as the attribution of nodes to processors is only determined by the parallel sparse matrix factorization heuristic. A better partitioning scheme should take also in consideration the load balance and the volume of data transmitted across the network for device evaluation and stamping. Also, the multiprocessor performance is limited when the interconnection network is saturated, as discussed in Section 3.1.2. A better scheduling heuristic could predict in advance the interface network saturation and replicate a subset of the tasks in different processors in order to alleviate network traffic. In spite of these shortcomings, the scheduling is quite effective for the Numerical Engine architecture described in Chapter 4. Empirical evidence based on simulated results at behavioral level and reasonable cost functions for data transmission and device evaluation in the Numerical Engine shows that the bus saturation for model evaluation and stamping typically occurs when the number of processors is around twenty. Since the target number of processors P is relatively small ($4 \leq P \leq 16$), it seemed unnecessary to write a more sophisticated partitioning and scheduling algorithm.

Figure 3-10 depicts in detail the data structures associated with the evaluation of non-linear devices and the stamping of the Jacobian matrix and right-hand-side vectors across processors. These data structures are created during the partition and scheduling precompilation phase, before the actual transient analysis loop starts.

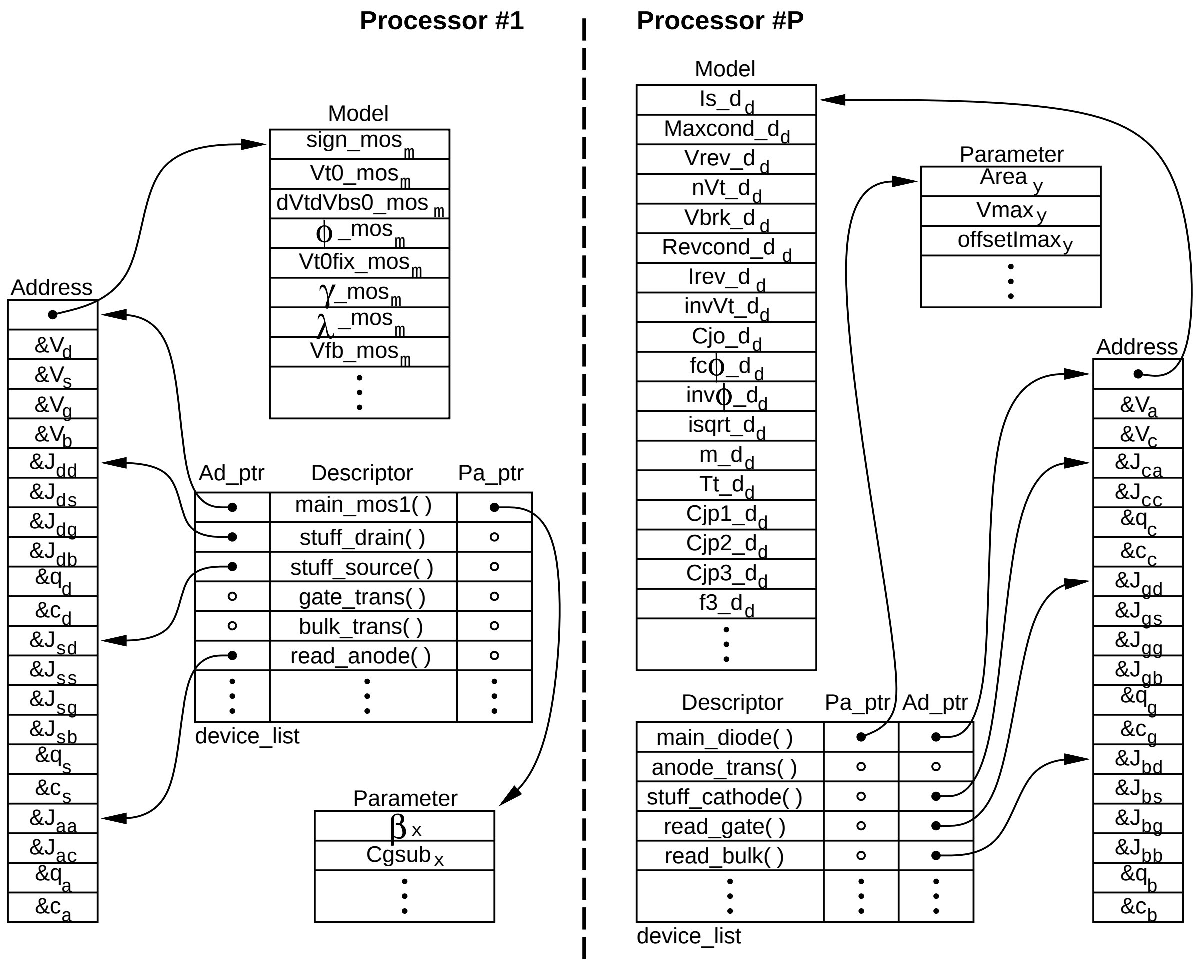


FIGURE 3-10: Distributed non-linear device data structures

Unlike the Linear Device and Voltage Source Table, which contains very few entries representing device *types*, the device_list structure for non-linear devices in each processor contains entries corresponding to device *instances*, as the stamping of each instance might be distributed across processors in different ways. Each entry in the *device_list* contains a descriptor that corresponds to a procedure call that is precompiled during the partitioning and scheduling of device evaluation and stamp tasks, a pointer Ad_ptr to a list of addresses related with the procedure execution, and a pointer Pa_ptr to a list of double precision floating point numbers used during the device evaluation. The procedure call could represent a device evaluation procedure main_xxx(), a local stamping procedure stuff_xxx(), a network transmission of the contributions xxx_trans(), or a procedure that reads device contributions from the network and stamps them locally read_xxx().

Additional data necessary for non-linear device evaluation is supplied by the Model structure depicted in Figure 3-10. It is common in integrated circuits that many transistors and diodes share a common set of parameters like oxide thickness, Fermi level, mobility, etc. Some operations involving these fundamental parameters can be computed before the execution of the transient analysis algorithm into a Model that is shared by many devices, thus saving execution time and memory space. For example, Figure 3-10 depicts the actual model parameters for a SIMLAB mos1 MOS transistor device present in processor #1, and the actual model parameters for a SIMLAB diode bipolar junction diode device present in processor #P. Since the space required by these models is typically very small (usually less than a thousand double precision floating point words), the precompilation scheme will assign a copy of all models to each processor.

Following the simulation steps related to Figure 3-10, processor #1 uses the local voltage vector information during the execution of main_mos1() to determine the drain, source, gate and substrate voltages of the particular transistor instance x depicted in Figure 3-10, along with its model (e.g. a long channel NMOS) parameters and instance parameters such as $\beta_x = \frac{\mu\epsilon\epsilon_0}{T_{ox}}\frac{W_x}{L_x}$ in order to compute all the contributions for the charges and current vectors in all nodes connected to the transistor, as well as the charge and current derivatives. Assuming the drain and source nodes are stored locally, stuff_drain() and stuff_source() will add the charge and current contributions q_d, c_d, q_s and c_s to the right-hand-side vectors, as well as their derivatives J_{d*} and J_{s*} to the Jacobian matrix. gate_trans() and bulk_trans() will transmit the contributions q_g, c_g, q_b, c_b, and their derivatives J_{g*} and J_{b*} to processor #P, which in turn will receive from the network these contributions during the execution of the read_gate() and read_bulk() procedures, and add them to the appropriate local entries in the right-hand-side vectors and in the Jacobian matrix. Similarly, processor #P will execute main_diode() and stuff_cathode() locally, and will use anode_trans() to transmit the anode contributions to processor #1, which in turn receives them and adds their contributions appropriately during the execution of read_anode().

3.3 Device Evaluation and Contribution Stamping Multiprocessor Performance

In order to simplify the scheduling of the device evaluation and the stamping operations in a multicomputer, as well as to obtain an estimate of the achievable speedup for a particular hardware configuration, one can represent the execution of the different procedures for device evaluation and stamping mentioned in Section 3.2.2 as the tasks or vertices of a task graph, and their interdependences as task graph edges, in the same fashion as described in Section 2.3 for the operations related with the sparse matrix decomposition.

Figure 3-11 shows two task graphs that represent the evaluation of the non-linear devices and the stamping process corresponding to the situation shown in Figure 3-10. Both cases represent the task graph after the assignment of tasks to processors, thus reflecting the introduction of new tasks reflecting either the need to replicate some tasks(a) or the need to transmit contributions across the network(b).

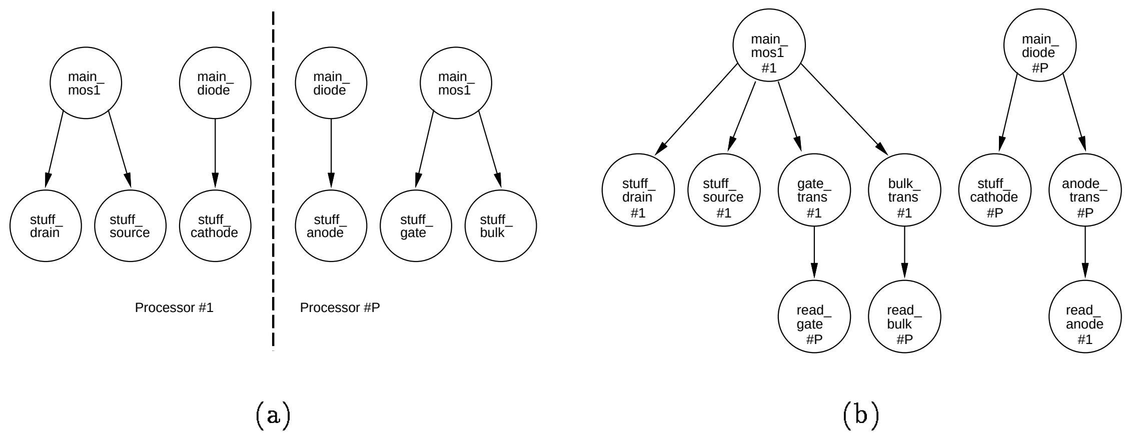


FIGURE 3-11: Task graphs associated with the execution of the non-linear device evaluation and stamping procedures with task replication (a) and with data transmission (b)

Figure 3-11(a) depicts the situation previously described in which absolutely no contributions are transmitted over the network, thus task main_mos1() is replicated in processor #P, as well as the main_diode() task is duplicated in processor #1. In this case, the graph associated with device evaluation and stamping tasks represents an easy parallel problem since all processors can work independently in as many independent tasks as the number of devices. The *height* (or number of levels H) of the graph is two. In spite of these advantages, experimental simulated data (Table 3-1) suggests that using this technique yields very poor multiprocessor utilization due to the task replication overhead. The network configuration in this case is irrelevant since no data is transmitted.

Figure 3-11(b) depicts the previously described situation in which the contributions of nonlinear devices that have terminal nodes stored in different processors are shipped across the network using the xxx_trans() and read_xxx() pair of procedures. Each vertex in the task graph in Figure 3-11(b) also shows the processor that will execute a particular task. Also in this case, the associated task graph has the desirable characteristics necessary to achieve a high degree of concurrency in a multiprocessor system. The graph is very wide, so balancing loads is simplified by the large number of independent tasks, and the height of the graph is at most three levels, which implies that the critical path is very short. Experimental simulated data (Table 3-1) for the Numerical Engine suggests that the usage of this technique yields good multiprocessor utilization for a moderate number of processors, thus it has been chosen for the actual behavioral level simulation of the execution of SIMLAB.

Table 3-1 shows the predicted number of Numerical Engine clock cycles required for the

Processors	Task Replication		Data Transmission		
	Cycles	Utilization	Cycles	Utilization	Bus Utilization
1	310,502	1.00	310,502	1.00	0.00
2	221,502	0.70	160,864	0.97	0.06
4	127,673	0.61	82,882	0.94	0.18
8	68,476	0.57	42,100	0.92	0.42
16	37,858	0.51	21,856	0.89	0.85

Table 3-1: Comparison of predicted Numerical Engine performance for device evaluation and stamping with task replication and with data transmission (shmem)

device evaluation and the contribution stamping of shmem, a static memory read test circuit that has a total of 1760 linear and non-linear devices. The cost functions for the device evaluation were generated by counting the total number of floating point instructions. If a particular device operates in different regions, an average value of the operation counts in the distinct modes was used. For instance, the mos1 transistor can operate in the *saturation*, *linear*, *accumulation* or *subthreshold* regions. The cost functions for data transmission reflect the actual number of elements that needed to be transmitted as well as the message overhead. In Table 3-1, the first column represents the number of processors. The second and third columns show respectively the number of clock cycles and utilization for the scheme that duplicates when necessary in multiple processors the evaluation of non-linear devices, while the fourth and fifth columns represent the results for the scheme that requires the transmission of non-linear device contributions across the network. The sixth column shows the bus utilization, indicating that the network will saturate when $P \approx 20$.

Assuming that the evaluation and stamping for non-linear devices account for most of the time spent, the results for total task replication indicate that even for two processors, roughly a third of the device evaluation tasks had to be replicated in another processor. Also, it seems that the work involved in the devices evaluation tasks tend to double as the number of processors increase, or in other words, as the number of processor grows, the average non-linear device tends to span across two processors, which is reasonably consistent with the characteristics of the shmem circuit. Many mos1 transistor devices will tend to have two out of four terminals (substrate and source) pinned to voltage sources, while the remaining that have three terminals connected to internal circuit nodes are compensated by the mosdiode contributions, which tend to always be local as the substrate of most transistors is connected to a voltage source, either Vdd or Gnd.

Even though the results listed in Table 3-1 for remote stamping without task replication for non-linear devices are quite good and undoubtly superior than the results for task replication, they are quite disappointing in the sense that the utilization results are similar to the sparse matrix results described in Chapter 2, while obtaining concurrency for device evaluation and stamping is clearly an easier problem than sparse matrix factorization. Aside from the obvious bus saturation problem previously discussed, that certainly did not happen in any of the results listed in Table 3-1 as the largest bus utilization was 85%, several factors contributed to these relatively poor results. These factors include the extra cost of the added xxx_trans() and read_xxx() tasks, poor load balance, and poor scheduling due to limited buffering.

The poor load balance is related to the fact that in the current version of SIMLAB the distribution of nodes to processors is solely dependent on the sparse matrix partitioning scheme. Even worse, according to the current device partitioning scheme, if all nodes that a particular device is connected to reside in the same processor, this processor will receive the entire load of that device evaluation and stamping, regardless of its previous load status. Probably, the load balancing is the single largest cause of performance degradation in Table 3-1.

The poor scheduling problem is related to the lack of buffering and consequently, is related to a severe restriction in the task scheduling freedom. In the current scheme, the device contributions computed in main_xxx() are kept in registers, so all tasks of the type stuff_xxx() and xxx_trans() related to the same instance must be executed immediately after main_xxx(). The actual task scheduling heuristic is also very simple, scheduling all mos1 devices first because they tend to send across the network most of the contributions, then mosdiode and diode devices are scheduled next as they tend to send fewer contributions across the network, followed by the evaluation of linear devices, which do not contribute at all for the network traffic. However, only an incremental improvement in performance for the Numerical Engine could be obtained if the scheduling restrictions are eliminated, as their role on the performance degradation is probably minimum, since the severe restrictions imposed in the scheduling freedom are diminished substantially by the extraordinary low height of the task graph.

Perhaps the greatest impact of buffering the contributions of the non-linear devices is in the extension of the methods discussed in this chapter to general purpose multiprocessor architectures. Most parallel computers available today have an extremely high message latency overhead, usually in the order of tens to hundreds of microseconds, or in the order of thousands of processor cycles. Aside from the fact that a small number of contributions to the same right-hand-side or Jacobian matrix entries do not need to be transmitted multiple times, lumping together all the messages from processor P_{orig} to processor P_{dest} to reduce the overhead represented by the message setup, so that a maximum of P^2 messages are transmitted across the interconnection network regardless of the size of the problem, will play a very significant role on the performance of the system.

Having studied in Chapter 2 how to use efficiently the computational resources of a specialized parallel computer for the sparse matrix solution, and in this chapter techniques to distribute amongst several processors the device evaluation and the stamping of their contributions to assemble the network equations, we intend to discuss in detail in the next chapter the proposed architecture of the Numerical Engine.

4

Numerical Engine Architecture

The Numerical Engine architecture supports the fast parallel factorization of sparse matrices using specialized hardware and the combination of scheduling and storage algorithms, as described in Chapter 1. The objective of the project was the design of a multicomputer containing processing elements with added hardware support for fast sparse *gaxpy* execution. *Gaxpys,* defined as $a_{ij} = a_{ij} + a_{ik} \times a_{kj}$, are the most frequent operations in matrix decomposition.

The Numerical Engine consists of a small number P $(4 \leq P \leq 16)$ of Processing Elements (PEs) interconnected through two high speed synchronous busses as shown in Figure 4-1. Physically, each PE will be contained in a removable printed circuit board (PCB), and is designed to operate at 50 MHz, while the backplane contains the properly terminated synchronous busses, designed to operate at 25 MHz. DBus is a 64-bit wide data bus and can be used to broadcast a double precision floating point word per clock cycle, corresponding to a total bandwidth of 200 Mbytes/s. IBus is a 32-bit wide data bus and can be used to broadcast either integer data or addresses in one cycle, corresponding to a total bandwidth of 100 Mbytes/s. A simple interface system connects the PEs with the host processor via the parallel SBus interface, which has a 32-bit wide data bus and an achievable bandwidth of 36 Mbytes/sec.

In order to achieve high speed for the sparse *gaxpys*, each PE has multiple interleaved memories to supply data at high rates for the floating point unit, support for the concurrent generation of addresses and writeback, and a simple but fast pipeline interlock system. Figure 2-15, in Chapter 2, depicts the dedicated datapath for fast O^2SA update, showing the essential components of each PE. In order to provide high speed control with the smallest possible latency for the various PE subsystems, the controller is a simple microprogrammed unit, containing a very wide microinstruction memory tightly coupled with each subsystem. This organization simplifies the control design and permits a very flexible utilization of the resources.

After power-up reset, the microcontroller is placed in a microprogram boot mode, loading the microinstruction memory for each PE from the host computer via the Sbus and the Ibus. Once the microprogram is loaded, execution is started, initializing all internal registers and

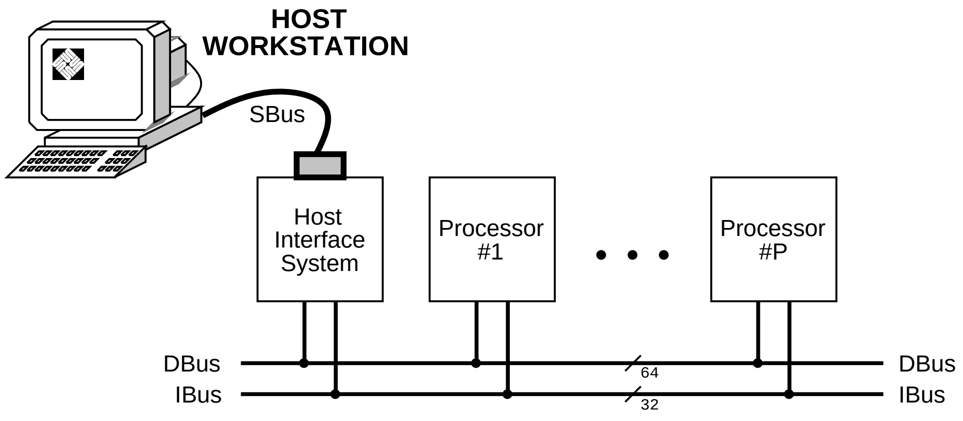


FIGURE 4-1: Overview of the numerical engine architecture

putting all PEs in a memory emulation mode that maps their internal memories into the virtual address space of the host. Also, under host control, the PEs can start the execution of a microprogram procedure. The advantage of this organization is that it greatly simplifies the hardware, and the system debugging, as a debugger program like *dbx* can ready access any internal PE memory as part of the virtual memory space of the host.

In the following sections, we will further describe the proposed architecture in detail. Section 4.1 will describe in detail system-wide considerations such as timing conventions and the general operation of the interconnection network. Section 4.2 will describe in detail the architecture of a single PE. Section 4.3 will discuss the emulation of the proposed architecture and some programming tools used to develop the emulator. Finally, Section 4.4 summarizes the performance results obtained in a detailed RTL simulation and compares them with the performance measured in general purpose computers.

4.1 System Considerations

4.1.1 Timing Conventions

An essential part of the system design is the set of timing rules, or timing conventions. Figure 4-2 depicts the basic clocking methodology. Each processor has its own local clock Clk with a 20 ns period. The rising edge of Clk clocks all PE registers.

The bus is driven by the BusClk signal, which is distributed to all processing elements and drive all bus input and output registers. BusClk period is 40 ns, and the bus data must be stable t_S (setup time) before the rising edge of BusClk and remain in the same state for at least t_H (hold time) after the clock transition in order to safely capture the data in the bus input registers. The skew between the rising edges of Clk and BusClk should be kept within a small percentage of the internal clock period.

Figure 4-2 also depicts the two types of signals issued by the control unit: the *lead* control signals come directly from the pins of the microprogram memory and are designed to be stable

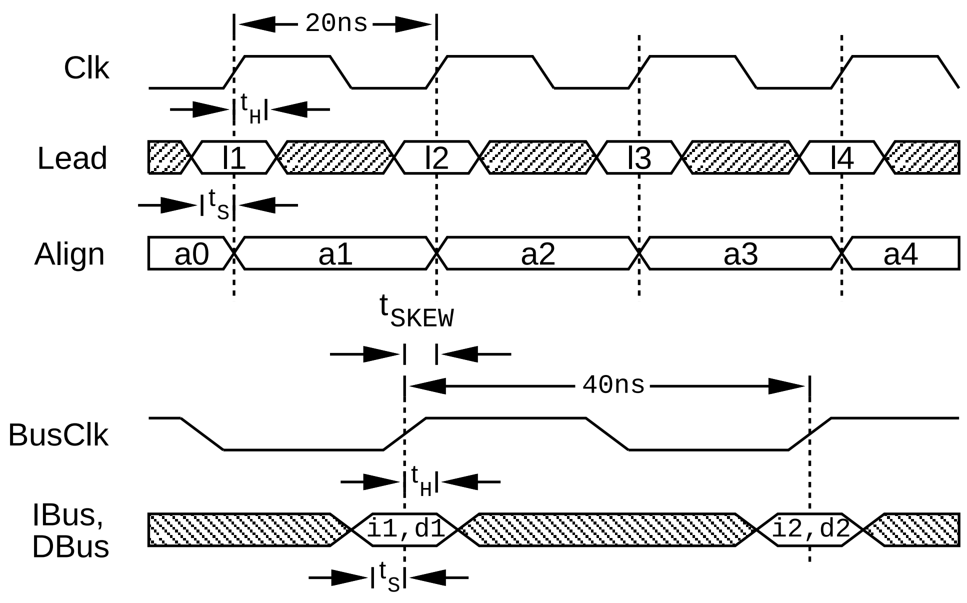


FIGURE 4-2: Timing conventions

$t_S \approx 4ns$ before the rising edge of Clk, and remain in the same state for at least $t_H \approx 2ns$ after the clock transition, and the *align* signals, which need a microinstruction register to keep the value stable throughout the entire cycle defined by Clk. *Lead* signals are used to specify the write address for the register file, the integer and floating point instructions that will be executed during the clock period starting at the rising edge of Clk, clock enable signals for datapath registers, finite state machine inputs for various PE subsystems. They also specify constants for the loop counter, and branch locations for the microprogram instruction pointer. *Align* control signals are used to enable various bus tri-state output drivers, specify read addresses in the register file, as generic input for slow combinational logic, or as control input for finite state machines that have tighter requirements on t_S than the usual. The advantages of using *Lead* signals, when possible, are to save hardware by avoiding extra microinstruction registers, and to reduce the overall latency in the execution of microinstructions.

A few registers, specially the microinstruction registers, need special phase synchronization with the Clk signal. For these critical cases, the scheme proposed in Figure 4-3 can be used to synchronize the output of a register or a logic gate with the input Clk signal. A dummy output signal line is added, having the same clock to output register delay as other outputs and a similar capacitive load. Inside the phase locked loop (PLL), the control voltage of the delay line will be self adjusted in such way that the total delay from the input Clk signal to the output dummy line is exactly one clock period. A specially interesting implementation of the PLL is the *Cypress 7B992 Programmable Skew Clock Buffer*. This device allows synchronization of multiple dummy lines or real output signals with the input clock or with programmable skewed versions of it.

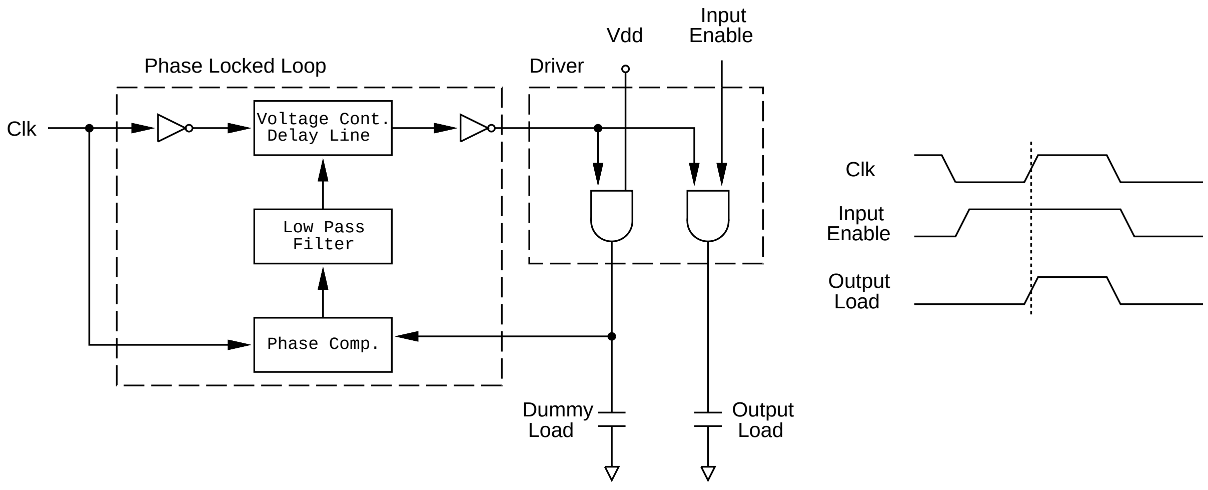


FIGURE 4-3: Synchronization scheme for aligning arbitrary outputs with the clock input signal

4.1.2 Interconnection Network

The interconnection network was designed keeping in mind the results predicted by the scheduler described in Chapter 2. A fast bus interconnect was a natural choice, as the ability to broadcast a normalized row to several processors plays a significant role in the sparse matrix decomposition.

A second consideration for choosing the network topology was the low latency associated with the bus, as the network latency affects directly the size of the critical path in the multitask graph (DAG) that represents the sparse matrix factorization process. Chapter 2 gives a detailed description of the parallel sparse matrix factorization and scheduling algorithms.

The usage of a bus for the interconnection network was recommended in face of the relatively small amount of concurrency that is possible to achieve during the sparse matrix factorization. According to scheduler predictions for infinite bandwidth, zero latency networks, as depicted in Figure 2-17, the maximum achievable speedup for the test matrices is in the neighborhood of $s_{max} \approx 40$. However, in order to keep a reasonably high level of processor utilization, a good number of processors would be $P \leq 20$. If more than $P = 16$ processing elements were required, a simple bus interconnect would become impractical, as the overall bus capacitance grows linearly with P and it becomes increasingly difficult to drive it at high speed.

Finally, the usage of a bus as network topology has other advantages like simple operation, and no need for complex routing algorithms. Also, it is easy to physically assemble the system, as each PE would be in a separate PCB card that could be inserted in the backplane containing the bus interconnect.

The goal for the bus throughput was also selected in face of the scheduler predicted results. As indicated in Figure 2-17, there is no substantial difference between the predicted performance for a system containing $P \approx 16$ processors and a bus able to broadcast a floating point data and its corresponding column address at the same rate as the *gaxpy* operations, and another one with the same number of processors and the bus operating at half that speed. By setting the target rate of 50 million *gaxpys* per second per processor, the corresponding demand for a $(64+32)$ bit wide data bus operating at 25 Mhz was considered reasonable, and does not seem to be very difficult to achieve.

Figure 4-4 depicts in detail the bus electrical interface, with emphasis on the output circuit for a generic signal from Processor #1 and the input circuit for that signal to Processor #P. All signals are both input from and output to the bus in all processing elements, except the Reset signal, which is driven by the host interface only and is an input in all PEs. The 25 Mhz BusClk signal is independently driven, and latches input data to the bus or output data from the bus on its rising edge. All other bus signals comply electrically with the Futurebus+ standard (IEEE P896.1). These signals feature registered open-collector output drivers with a series Schottky diode to reduce the capacitive loading to the bus. These drivers and receivers are physically placed in each processing element board very close to the bus in order to keep the length of the stubs to a minimum, thus reducing the noise introduced by the reflection. A 2-V pullup is used to terminate the bus in both ends, causing a BTL logic level swing of approximately 1-V, which considerably reduces the power necessary to drive the bus load capacitance. The receiver input is a differential stage, comparing the bus voltage with an internally generated voltage reference. Using bus interface devices like the *Texas Instruments SN74BCT979*, shown in Figure 4-4, placed physically very close to the bus connector in each PE, it is possible reduce the total propagation delay in the bus to meet the speed requirements. These devices are able to drive DC loads as low as 10 Ω, and with proper bus design it is possible to obtain input switching in the incident wave for very high speed operation.

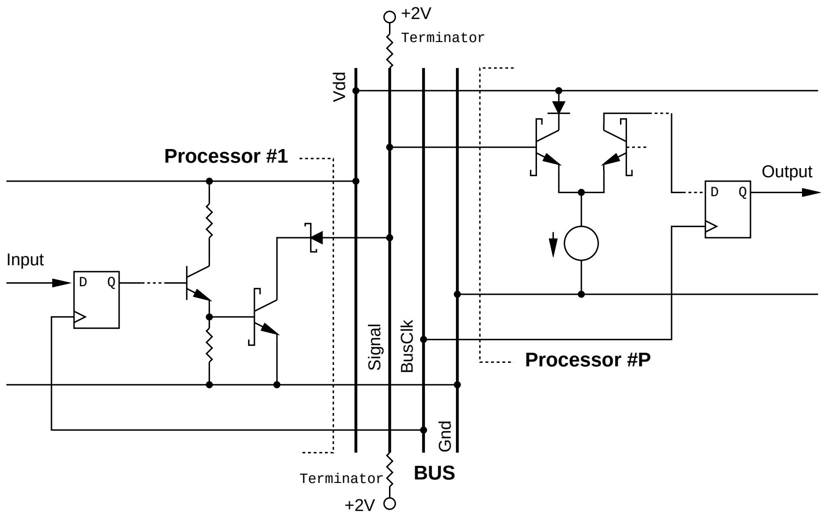


FIGURE 4-4: A generic bus signal input and output circuits

Figure 4-5 shows all the bus interface signals. They can be classified into four major groups, DBus, IBus, QuietBus, and WiredBus. DBus is used for broadcasting 64-bit double precision floating point data, and is connected to the floating point unit in each processor via a bidirectional FIFO. IBus is used to broadcast 32-bit integer or address data, and is connected to the QBus of each PE through a bidirectional FIFO. The internal architecture of the processing elements will be further discussed in Section 4.2. Since all bus transactions are pre-scheduled during the program load time, the handshake is done by only two signals in the QuietBus, which are issued by the processor that has the bus ownership. The `Valid` signal indicates that the data placed on the DBus and on the IBus is valid, while the `Done` signal indicates that next scheduled processor can take control over the bus in the next cycle.

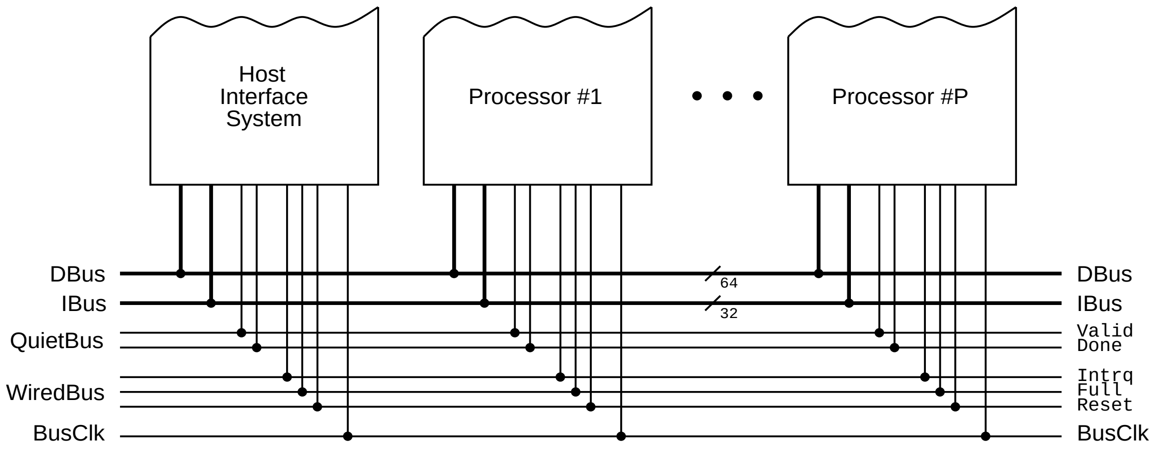


FIGURE 4-5: Bus interface signals

While signals in the three groups previously described can only be asserted by the processing element (or the host interface system) that has control over the bus, the "emergency" signals in the WiredBus can be asynchronously asserted by any processor, as these signals form a *wired-or,* exploiting the open-collector output structure of the bus interface drivers. The `Full` signal can be asserted by any processor whose FIFO receiving data from the bus is almost full. The processor that has control over the bus monitors the `Full` signal and has to immediately stop sending valid data to the others processors, entering in a *bus idle state.* This state persists until the `Full` signal is de-asserted. This mechanism automatically slows down the bus when some of the processing elements are consuming more time than predicted to perform internal tasks. A good scheduling algorithm should be able to minimize the number of bus idle cycles. A watchdog can be set in the Host Interface System to monitor this signal and take some action in the case no progress is made after a specified amount of time. A special care must be taken by the scheduling algorithm to insure that a processor will never try to send any data that might overflow the FIFO that outputs data to the bus before making sure that the FIFO that inputs data from the bus will not overflow, which is the only possible source of bus deadlock. The deadlock analysis is simplified because only the amount of transmitted data and the FIFO sizes are relevant, and not the actual timing. The Intrq signal is issued by a processor to indicate an error situation that cannot be coped with by the normal processing flow. Upon receiving an Intrq signal, all processors will execute a microcode trap that usually involve cleaning up all bus FIFOs and going back to the initial memory-mapped emulation state. The Reset signal can only be issued by the host interface, and forces all processors to clean all bus FIFOs and entering in the microprogram boot mode, which is discussed in detail later. The Reset signal is always asserted after power-on.

Figure 4-6 helps to clarify the bus interface protocol, describing in the detail the passage of bus control from processor #1 to processor #P. During the first three cycles in Figure 4-6, PE #1 has control over the bus and is transmitting message n, while during the last cycle PE #P has control over the bus, starting message $n+1$. At $t = 40ns$, PE #1 does not have any valid data available in the output FIFO and signals that state to the other processors by de-asserting the Valid signal. The other processing elements will then ignore any data present in DBus and IBus. In the next cycle, at $t = 80ns$, processor #1 will place in IBus and DBus the last piece of data belonging to message n, signaling to the other processors by asserting the Done line. When Done is asserted, all other processors must watch the bus precisely in the following cycle. In the next cycle, at $t = 120ns$, IBus holds the address of the processor that will take control over the bus for message $n+2$, or srcpr(n+2), and DBus holds a 64-bit boolean vector that contains a bit set for each processor that will be a recipient of message $n+1$, or destvec(n+1). We shall refer to the $\{\text{srcpr}(n+2), \text{dstvec}(n+1)\}$ data pair as the *message control vector*. Processor #P can start driving the bus with the first piece of data for message $n+1$ at $t = 120 + t_H$, exactly one cycle after the Done signal was detected, because it learned, in the end of message $n-1$, that $(\text{srcpr}(n+1) = P)$, which informed it to take control over the bus once message n was finished.

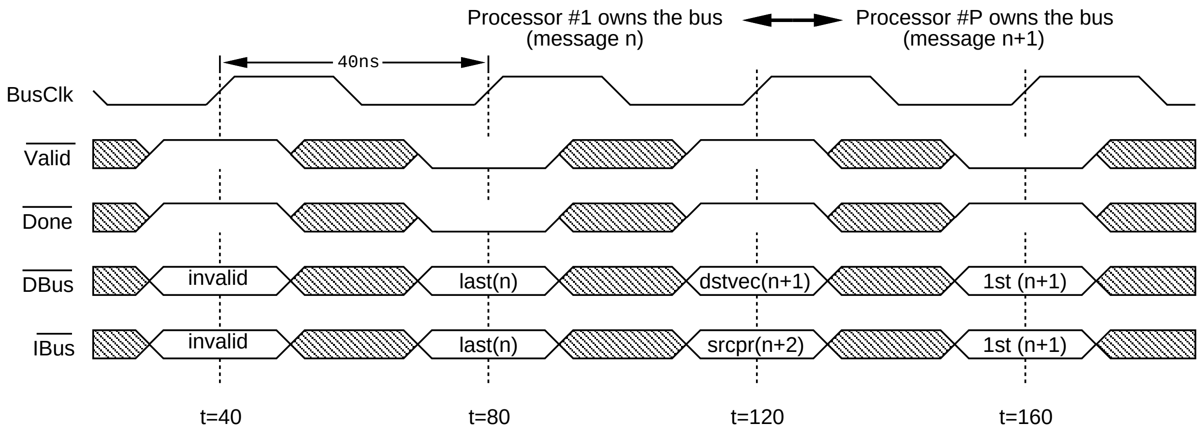


FIGURE 4-6: Bus protocol for passing control from processor #1 to processor #P

This simple interface mechanism allows messages to be *selectively* sent to up to 64 processors and at the same time provides a low overhead in passing the control from one processor to another. These characteristics are very important for sparse matrix computations, as most messages are very small and transmitted typically to few processors. Using this scheme, only a single cycle of overhead per message is needed.

Figure 4-7 depicts the bus activity during the microcode boot. In order to load n microcode words, each 128-bits wide, $4n+2$ valid bus cycles are necessary. Upon receiving the Reset signal, all processors will flush all their FIFOs and enter in a microcode load mode. The first valid 32-bit word sent in the IBus (LAddr) specifies the starting load address. After this word, each group of four valid IBus words (128 bits) is loaded in sequential addresses of the microprogram memory, starting from LAddr. The last microcode word (n-th) is executed after the Reset signal is de-asserted. Typically, the last microcode sent through the IBus contains a datapath NOP (no operation) and a branch to the starting address of the microprogram (XAddr). Microprogram execution is immediately started, and typically puts all the processing elements in a memory mapping emulation mode, as described earlier.

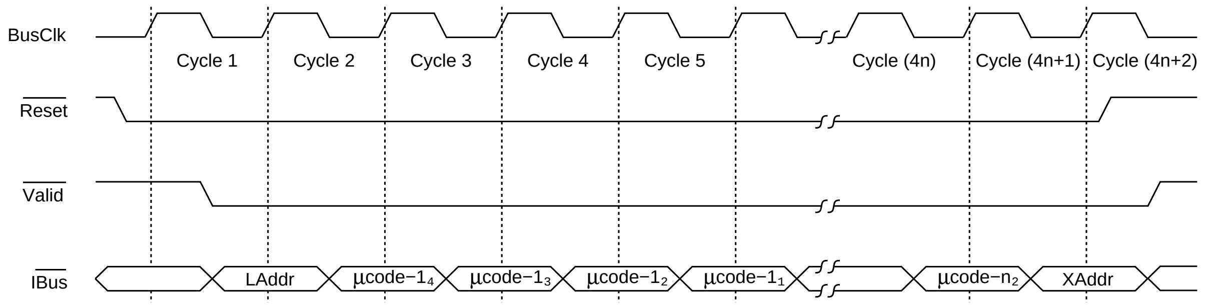


FIGURE 4-7: Microprogram boot sequence after reset

The next section is a detailed description of the internal architecture of a processing element (PE).

4.2 Processing Element Architecture

4.2.1 Overview

Each processing element contains a floating point unit and register file, a special purpose memory system that is tuned for the fast access to sparse matrix data, an external bus interface, and the microprogrammed control. These blocks are interconnected by five internal busses: HBus, QBus, XBus, YBus, and ZBus. The general block diagram of each processing element is shown in Figure 4-8, the thicker lines representing 64-bit wide paths.

The system is designed to operate with a *system clock* of 50 Mhz. The static memories, the microcontroller, the address generators, the short internal busses, the floating point unit, and the register file are designed to operate with a 20ns period. The external bus and the dynamic memories, which are the slowest components of the system, operate with a 40ns period. The

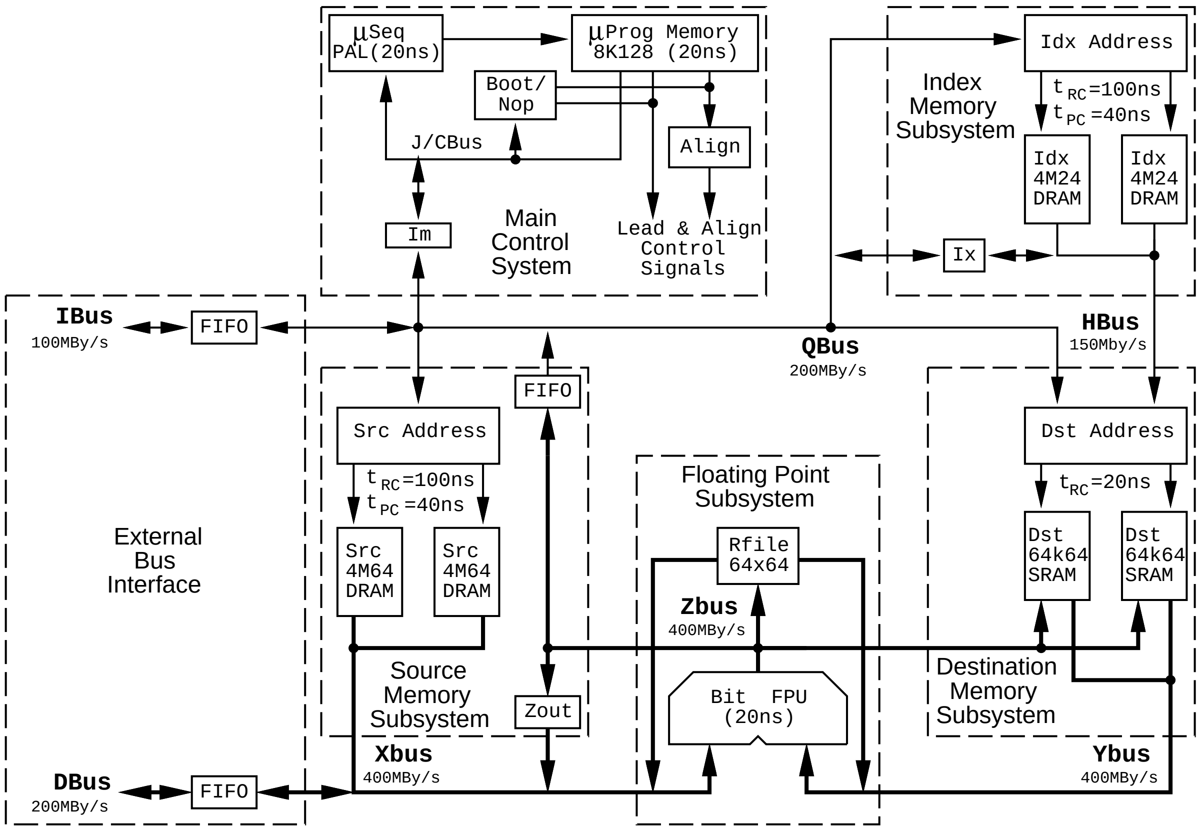


FIGURE 4-8: Processing element internal architecture

proposed clock cycle is rather aggressive, especially because it is difficult to obtain accurate timing simulations, but not impossible to achieve using the technology available today.

The floating point unit (FPU) is a *Bipolar Integrated Technology, Inc. BIT 2130* FPU operating at 50Mhz. It contains an independently-controlled floating point multiplier, ALU, and divide/square root unit. The FPU is connected to three, physically very short, high speed busses (XBus, YBus and ZBus) that have a collective bandwidth of 1.2 Gbytes/second (50Mhz $\times$ 3 ports $\times$ 8 bytes per port). In each clock cycle, two operands can be read from XBus and YBus and one can be written to the ZBus in order to achieve the maximum utilization of the FPU pipeline: 100 MFlops. The floating point register file holds general purpose temporary floating point data that can be immediately accessed, as well as important floating point constants, such as 1, 2, $\log_2 e$, and $\frac{2}{\pi}$. It consists of a three port 64-word by 64-bit wide register file with a cycle time of 20ns. The register file outputs are connected to the XBus and the YBus, while its input is connected to the ZBus. Using this scheme, the register file can match the FPU bandwidth of 1.2 Gbytes/second. The floating point unit and the register file will be discussed in detail in Section 4.2.3.

The special purpose memory system is designed to provide fast access to sparse matrix data in order to keep the floating point pipeline full. There are three major blocks that compose the memory system: the destination, source, and index memories. These three subsystems will be discussed in more detail in Section 4.2.4.

The *destination memory* subsystem holds a set of destination rows for a row-update operation, targets for gather-normalize operations, right hand side vectors, voltage and current vectors, device intrinsic parameters such as Vt_0 for all NMOS devices, floating point constants needed in the evaluation of transcendental functions, and small amounts of generic temporary data. When the data corresponds to the targets for row-update operations, it is stored according to the O^2SA scheme, as discussed in Chapter 2. The destination memory subsystem contains a two-way interleaved 128K-word by 64-bit wide separate I/O static RAM array with a cycle time $t_{RC} = 20ns$ and a special purpose destination address unit. Output data pins from the memory array are directly connected to the YBus, and input data pins to the memory array are directly connected to the ZBus, thus reducing significantly the latency of memory read-operate-write instructions. During the row update operation, in each clock cycle (20ns) the special purpose destination address unit receives a 24-bit column index from the HBus and adds it with the offset provided by the QBus in the beginning of the row update operation in order to compute the target element address in the destination memory. Combined with the address previously stored for memory writeback, the address generator provides the static RAM array two addresses every 20ns, typically one for a read operation in one memory bank and the other for a writeback operation in the other memory bank. This capability is necessary to match the bandwidth of the floating point pipeline via the YBus and the ZBus, up to 800 Mbytes/second. The destination memory latency is three clock cycles or 60ns, a period starting when the address is latched in the Dst Address block from QBus or HBus, and finishing when the corresponding read data is available at YBus. If a memory bank conflict occurs because a read or write operation is trying to access the same memory bank that the writeback operation needs, the memory destination controller will postpone the read or write operation, execute the high priority memory writeback and request a stall cycle from the main control. The processor stall scheme is discussed in detail in Section 4.2.2.

The *source memory* subsystem holds source rows for update operations in sequential addresses, the task descriptors that represent the instructions in the INST array in the Algorithm 2.6, and device instance parameters as described in Chapter 3, as well as general purpose large volume data. The source memory subsystem consists of a two-way interleaved 8M-word by 64-bit wide common I/O dynamic RAM array with a fast page mode cycle time $t_{PC} = 40ns$ and random access cycle of $t_{RC} = 100ns$, a 512-word by 64-bit wide descriptor FIFO, an address unit, and the Zout register. Data is read from the DRAM array into the XBus. The memory I/O data pins can also be used as inputs for loading the contents of the Zout register, which in turn can be independently loaded from the ZBus. The address unit is capable of loading an address from the QBus or incrementing for sequential data access, providing one address per clock cycle (20ns) for the interleaved memory array. The FIFO can hold up to 512 task descriptors that are sequentially read from the source memory in a single burst. The task descriptor data is sequentially accessed via the QBus. Every 20ns the source memory subsystem can either supply to, or receive from the XBus one 64-bit word. This capability is necessary to match the bandwidth of the floating point pipeline via the XBus, up to 400 Mbytes/second. The source memory latency is three clock cycles or 60ns, measured from the time the address is loaded from the QBus to the time that the appropriate data is delivered to the XBus. If the requested data is not readily available because it resides in a different memory page or hits the wrong interleaved data bank, the source memory subsystem will stall the processor until the proper data becomes available.

The index memory subsystem holds column indices for row update and normalize operations in sequential addresses, voltage vector indices, and general purpose memory pointers. The index memory subsystem consists of a two-way interleaved 8M-word by 24-bit wide common I/O dynamic RAM array with a fast page mode cycle time of $t_{PC} = 40ns$ and random access time of $t_{RC} = 100ns$, edge-triggered tri-state Ix I/O transceivers, and an address unit. The address unit is capable of loading an address from the QBus or incrementing for sequential data access, providing one address per clock cycle (20ns) for the interleaved memory array. Every 20ns the index memory subsystem can supply to the HBus one 24-bit word in order to match the requirements of the destination address generation unit during a row-update operation. The memory can either supply one 24-bit word to, or receive data from the QBus via a set of registered tri-state transceivers every 20 ns, matching the bandwidth required by the destination memory address generator, up to 150 Mbytes/second. The memory latency is two clock cycles or 40ns, measured from the time the address is loaded from the QBus to the time that the appropriate data is delivered to the HBus and latched either in the Ix registers or in the Dst Address input register. The latency is smaller than the source memory latency because the index address generator has to drive only a third of the capacitive load. If the requested data is not readily available because it resides in a different memory page, the index memory subsystem will stall the processor until the data becomes available.

The bus interface controls the data flow between the processing element and the external DBus and IBus. The datapath contains two bidirectional FIFOs, and a set of registered bus transceivers. The DBus connects to a 512-word by 64-bit wide bidirectional FIFO that buffers the external data to and from the XBus. Depending on the FIFO's t_S, t_H and t_{CO} characteristics, it might become necessary to only allow data be loaded in the FIFO from the Zout register and the output data from the FIFO might only be loaded in the FPU input registers. The direct exchange of data from the FIFO to or from the DRAM memories will only be possible if a FIFO with fast I/O interface is available. The IBus connects to a 512-word by 33-bit wide bidirectional FIFO that buffers the external data to and from the QBus. The bus interface controller can slow down the interconnection network in the case the part of the FIFO that receives external data becomes full. The bus interface controller will stall the processor when it attempts to read data from an empty FIFO, or in the case the processor tries to write data to a full FIFO. The system software is responsible for avoiding bus deadlocks. A detailed description of the network interface is given in Section 4.2.5.

The main control consists of a microprogram address register with sequencer logic, a 8K-word by 128-bit wide microprogram memory, the Align registers, the Boot/Nop registers, and the Im registered transceivers. The sequencer receives instructions, status information and jump addresses, generating a sequence of microaddresses for the microprogram memory. The sequencer operates with a 20ns period. The microprogram memory feeds back the sequencer with next state information, supplies immediate data to the QBus via the Im register, specifies loop counts, and also contains microinstruction fields that control all the subsystems previously described. A detailed description of the main control is given in Section 4.2.6.

Each processing element contains two hundred and twelve ICs and has an estimated power dissipation of 150 Watts. The following section presents a detailed description of the pipelined operation and some stall issues.

4.2.2 Pipelined Operation and Stall Issues

Let us consider the sparse matrix row update $i \to k$ that consists of several *gaxpys* of the form $a_{ij} = a_{ij} - a_{ik} \times a_{kj}$. Figure 4-9 depicts the relevant pipelined execution and stall issues of the tail end of one short row update, immediately followed by the head of a very long row update. Figure 4-9 highlights the sequencing aspects of the operation, instead of providing an accurate timing information. The windows in which the data is stable are actually much smaller than the depicted regions on Figure 4-9. In order to simplify the analysis, only stall requests generated by the three memory subsystems are taken in account. The analysis for stall requests from the external bus interface is analogous.

Upon receiving any of the stall requests stallreq_idx, stallreq_src, or stallreq_dst, the main control unit issues the stall signal, which forces a programmable no operation (nop) in all *lead* microinstruction signals, and holds the last command issued in the *align* microinstruction signals. Each subsystem will take different actions upon receiving the stall signal. We should refer to the cycles in which the main control issues the stall signal as *processor stall cycles.* All other cycles are referred to as *valid processor cycles.* Specifying all latencies in terms of valid clock cycles simplifies the control unit design and the overall system programming, as it makes possible to predict in advance which parts of a particular computation are being executed in the different units at every valid clock cycle.

The source-row column indices $(j - k)$, corresponding to a_{kj}, are read into HBus from the index memory subsystem, and added to the address of the element a_{ik} in order to compute the address of target element a_{ij} in the O^2SA array. In Figure 4-9, the index memory address of the first non-zero element a_{kj} after the diagonal is si2, loaded from the QBus at the end of clock cycle 1. Assuming that this memory request is located in a different memory page,

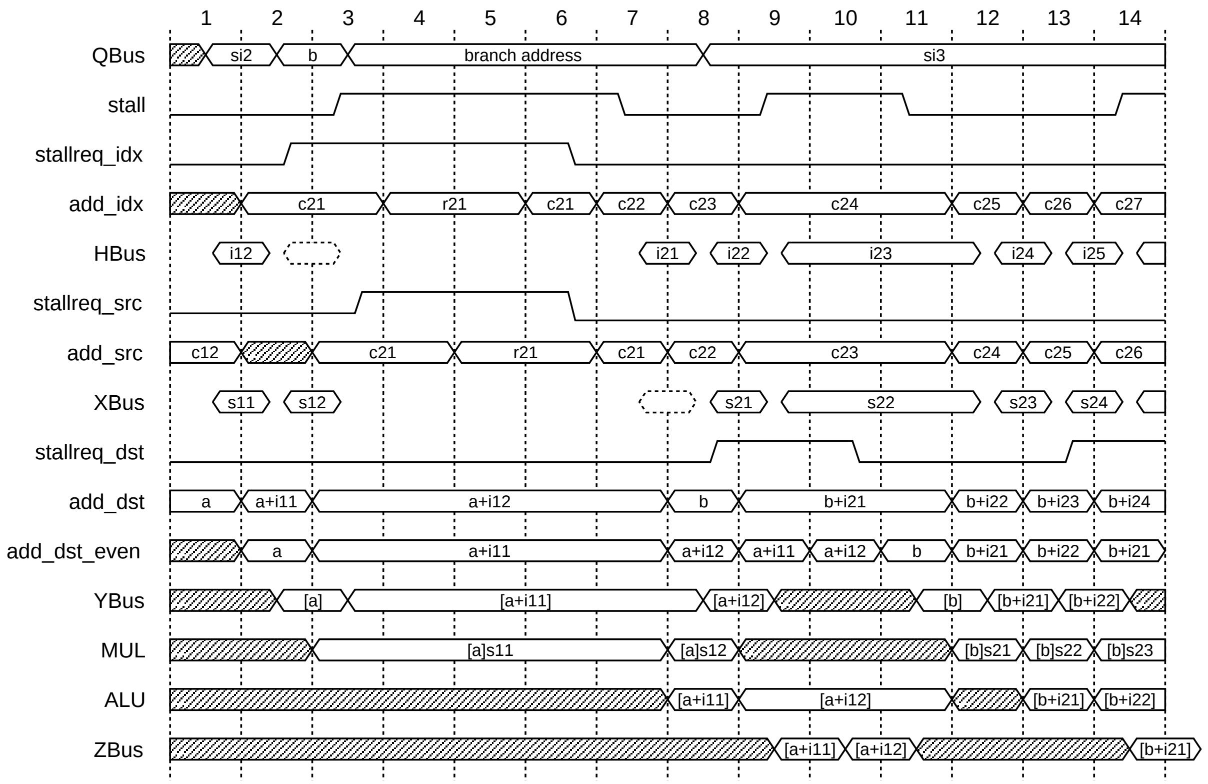


FIGURE 4-9: Processor element pipelined operation and stall issues

the index memory controller will detect this situation and request, using the stallreq_idx signal, a sequence of processor stall cycles. The first column index, labeled i21, will be available in HBus at the end of clock cycle 7. The next column indices in that row, labeled i22, i23, etc. will be available in the HBus at the end of each valid clock cycle, always two valid clock cycles after the corresponding request. Since cycles 9 and 10 are processor stall cycles, the index memory controller will hold the value i23 in the HBus, by keeping the $\overline{\text{cas}}$ signal active. At the end of clock cycle 1, the last piece of index data from the tail end of a short row update operation (i12), is present in HBus. The next cycle corresponds to a index memory subsystem nop operation, always necessary between two row updates, as there is always one more read data access (multiplier a_{ik}) in the destination memory than in the other memory subsystems per row update operation. Section 4.2.4 presents a complete description of the index memory subsystem.

The actions in the source memory subsystem in order to retrieve the actual numerical values of a_{kj} are similar to the index memory subsystem. The major difference is that the source memory requires one extra latency cycle since the address generator must drive a much larger capacitance in the DRAM array. Thus, all output data to the XBus is delayed by one valid clock cycle relative to the HBus. Since clock cycle 3 corresponds to a processor stall cycle, the source memory controller postpones the execution of the nop instruction until clock cycle

7, driving dummy data out, while during clock cycle 8 the first element, labeled s21, is placed on the XBus. This scheme, which helps reducing the number of requested stall cycles when nop operations are pending is discussed in detail in Section 4.2.4. In order to properly match the combined latency of the index memory and destination memory with latency of the source memory, it is necessary to delay source data by one valid clock cycle, using an internal floating point unit register. For example, the load of source data s21 in the floating point multiplier MUL is only done at the end of clock cycle 11. Section 4.2.4 presents a complete description of the source memory subsystem.

In Figure 4-9, the destination memory address of the sparse matrix row update multiplier a_{ik} is labeled b, and it is loaded from the QBus at the end of clock cycle 2. The first destination memory access in a row update uses the pass operator to generate the address, which appears in add_dst register in the next valid clock cycle, 8. Only a memory read is required for this access. All the other addresses, generated with the add operator, are of the form b+i21, b+i22, etc., corresponding to the sum of the address of the multiplier, a_{ik}, with the column index $(j - k)$ to obtain the address of a_{ij}. The latter require a read from memory, operate, and result writeback operations. Memory writeback operations cannot be stopped in the sense that once the data has been read during a valid processor cycle from the YBus, the proper result must placed in ZBus within two clock cycles, and the destination memory system must perform the writeback, corresponding to a self-drained pipelined operation. As a consequence, the floating point output data register does not follow the stall rules: it can be loaded even during a processor stall cycle for a writeback operation, using as enable a special signal generated by the destination memory subsystem. Also, in the case there is a memory bank conflict between the writeback and a read or a write operation in the destination memory, the writeback has higher priority and is executed immediately, while the stallreq_dst line is used to request a processor stall cycle, thus postponing the other memory operation. Figure 4-9 depicts the worst case in respect to interleaved memory bank conflicts: all destination memory operations use the even memory data bank. For example, during clock cycles 9 and 10, data is written in the destination memory addresses a+i11 and a+i12, both in the even memory bank. These writebacks finish the execution of the previous row operation and request a processor stall to postpone the read of the multiplier value $a_{ik} =$ [b], which is trying to use the same memory resource as the writebacks. Once the writebacks are finished, data corresponding to the multiplier [b] is placed in YBus at the end of clock cycle 11. A good scheduling algorithm should try to avoid these situations, choosing a row update operation that contains a multiplier in the odd memory bank. Scheduling algorithms are discussed in detail in Section 2.3.

An important pipelined operation issue is the *operand bypass detection*. This detection was left to the system software, and it is discussed in detail along with the destination memory subsystem specification in Section 4.2.4.

Assuming that b+i21, b+i22, and b+i23 correspond to even memory addresses, the writebacks of [b+i21] and [b+i22] to the even memory bank will also postpone the execution of the read from address b+i23. This situation could be avoided in many cases by reordering the elements after the diagonal of a source row for efficient interleaving. The impact of this reordering in the single processor performance was discussed in detail in Section 2.2.1.

Each of the processing element subsystems is discussed in detail in the following sections.

4.2.3 Floating Point Subsystem

The floating point subsystem consists of the floating point unit and the floating point register file. All the operands for floating point operations are double precision floating point numbers that conform with the ANSI/IEEE 754 standard. The floating point unit also supports a wide variety single precision floating point, 32-bit and 64-bit integer operations. The current version of the emulator software supports only double precision floating point instructions and 32-bit integer operations.

As discussed in Chapter 2, the floating point unit (FPU) should contain as many pipeline stages as possible to increase the FPU throughput. In that discussion, we have shown that the amount of concurrency for solving sparse matrices is not very sensitive to the number of pipeline stages in the floating point unit. However, there are commercially available FPUs that can operate at such high frequencies that demand data rates substantially higher than achievable with two-way interleaved memory systems using commercially available high density memories. Also, the design of a highly pipelined FPU is beyond the scope of this thesis.

In selecting a commercially available FPU, the units taken in consideration were the *Bipolar Integrated Technology, Inc. BIT 2130* and the *Weitek 3364*. The *BIT* part was chosen for several reasons:

- For the same bus clock rate, the *BIT* part has twice the bandwidth, since it has twice as many I/O pins. The Weitek part has three 32-*BIT* ports and needs multiplexing and buffering for accessing 64 *BIT* floating point data. The BIT part doesn't require any multiplexing, providing three 64-bit ports.

- The setup and hold times for the *BIT* part are much smaller, thus eliminating complex I/O buffering and reducing the memory latency.

- The *BIT* part supports multiple shifts for integer operands, which is very important in the implementation of transcendental functions such as e^x.

- Regarding flowthrough, the *BIT* part provides 20ns latency and 10ns throughput for ALU and MPY operations, resulting in up to 200 MFlops for vector operations and 100 MFlops for scalar operations. Since it is very difficult to operate the system with a 10ns clock period, we decided to use the *BIT* part in the scalar mode, with a clock period of 20ns.

The Weitek part presents 100ns latency and 50ns throughput, resulting in only 40 MFlops for vector operations.

- The *BIT* part DIV and SQRT times are 250ns and 450ns respectively. The Weitek part DIV and SQRT times are 850ns and 1500ns respectively. For many computations, the DIV and SQRT times are a limiting factor in the overall execution time.

The internal architecture of the *BIT* part and the microinstructions specified for its control are depicted in Figure 4-10. All internal registers of the *BIT* part are clocked by the Clk signal. The functional blocks of the *BIT* 2130 include an arithmetic and logic unit ALU, a multiplier MUL, a divide/square root DIV/SQRT element, internal datapath multiplexers, and temporary storage registers. Externally the floating point unit connects to three 64-bit wide busses, XBus, YBus, and ZBus, the various microprogram control signals, the $\overline{\text{Zen}}$ signal, and the flags. The $\overline{\text{Zen}}$ signal, active low, is generated by the destination memory subsystem to enable the Z output register load. In normal operation, load is enabled except in the case of a stall. The destination memory controller might override this situation in the case of a memory writeback, which cannot be stalled, as described in Section 4.2.2. Only two flags from the FPU are supported in the current implementation of the architecture emulation, FlagN, indicating a result smaller than zero, and FlagZ, indicating a result equal to zero.

The ALU is a combinational circuit that performs integer and floating point addition, subtraction, conversion, shift, compare and other operations. The ALU is controlled by the bitalu_l microinstruction. bitalu_l is a *lead* control signal, and it specifies:

`Axsel` Multiplexer select for the ALU input operand X. Selects between one of the four sources:

00	XBus
01	Areg
10	ALU feedback
11	MUL/DIV/SQRT feedback

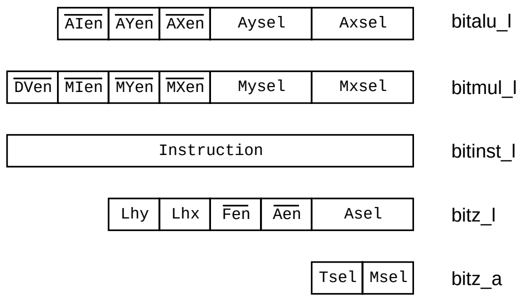
`Aysel` Multiplexer select for the ALU input operand Y. Selects between one of the four sources:

00	YBus
01	Areg
10	ALU feedback
11	MUL/DIV/SQRT feedback

$\overline{\texttt{AXen}}$ Active low, enables the Ax operand register load.

$\overline{\texttt{AYen}}$ Active low, enables the Ay operand register load.

$\overline{\texttt{AIen}}$ Active low, enables the ALU instruction Aop register load from the bitinst_l microinstruction.



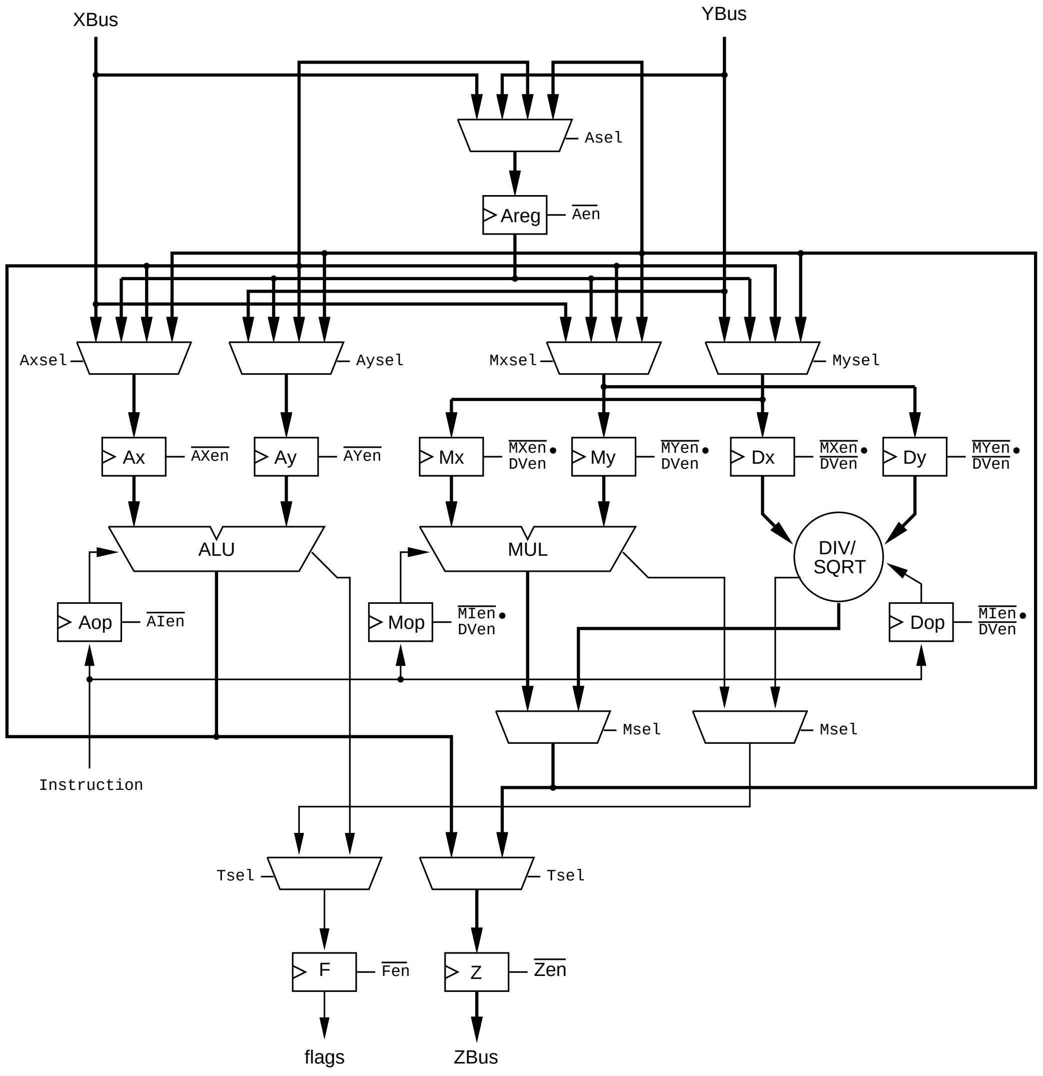


FIGURE 4-10: Floating point unit internal architecture

The multiplier is a combinational circuit that performs floating point and integer multiplications. The DIV/SQRT is a synchronous finite state machine that executes division and square root operations. Both units are controlled by the bitmul_l microinstruction. bitmul_l is a *lead* signal, and it specifies:

`Mxsel` Multiplexer select for the MPY/DIV input operand X. Selects between one of the four sources:

00	XBus
01	Areg
10	ALU feedback
11	MUL/DIV/SQRT feedback

`Mysel` Multiplexer select for the MPY/DIV input operand Y. Selects between one of the four sources:

00	YBus
01	Areg
10	ALU feedback
11	MUL/DIV/SQRT feedback

$\overline{\texttt{MXen}}$ Active low, enables the MPY/DIV X (either Mx or Dx) operand register load.

$\overline{\texttt{MYen}}$ Active low, enables the MPY/DIV Y (either My or Dy) operand register load.

$\overline{\texttt{MIen}}$ When active, enables the MPY/DIV instruction (either Mop or Dop) register load.

$\overline{\texttt{DVen}}$ When low, $\overline{\texttt{MXen}}$, $\overline{\texttt{MYen}}$, and $\overline{\texttt{MIen}}$ signals are used to individually enable the DIV/SQRT Dx, Dy and instruction registers Dop, while the corresponding MPY registers are disabled. When high, the DIV/SQRT registers are disabled, and the control signals enable the multiplier's Mx, My and Mop registers.

bitinst_l is a *lead* control signal that specifies the instruction executed by the internal computational elements. The value of bitinst_l can be internally stored in the Aop, Mop, and Dop registers, depending on the status of the $\overline{\texttt{AIen}}$, $\overline{\texttt{MIen}}$, and $\overline{\texttt{DVen}}$ control signals. A wide variety of 32-bit and 64-bit floating point and integer operations are available. The *BIT B2130* literature has a complete list of the instructions [Bit90].

bitz_l is a *lead* control signal, which controls the Areg and flag output registers. This microinstruction specifies:

`Asel` Multiplexer select for the Areg register input. Selects between one of the four sources:

00	XBus
01	YBus
10	ALU feedback
11	MUL/DIV/SQRT feedback

$\overline{\texttt{Aen}}$ Active low, enables the Areg register load.

$\overline{\texttt{Fen}}$ Active low, enables the flag output register F load.

Lhx When high, copy the XBus operand LSW to its MSW. Internally, the MSW is used for single precision operations.

Lhy When high, copy the YBus operand LSW to its MSW. Internally, the MSW is used for single precision operations.

bitz_a is an *align* control signal, which controls the M- and T- multiplexers. This microinstructions specifies:

Msel Multiplexer select for the output result and flags from the multiplier MUL and the DIV/SQRT unit. When low, selects the multiplier output result and flags; otherwise selects the DIV/SQRT unit output result and flags.

Tsel Multiplexer select for the Z and F output registers. When low, selects the ALU output and flags as inputs; otherwise selects the operands selected by Msel.

Figure 4-11 shows the internal structure of the floating point register file and the microinstructions specified for its control.

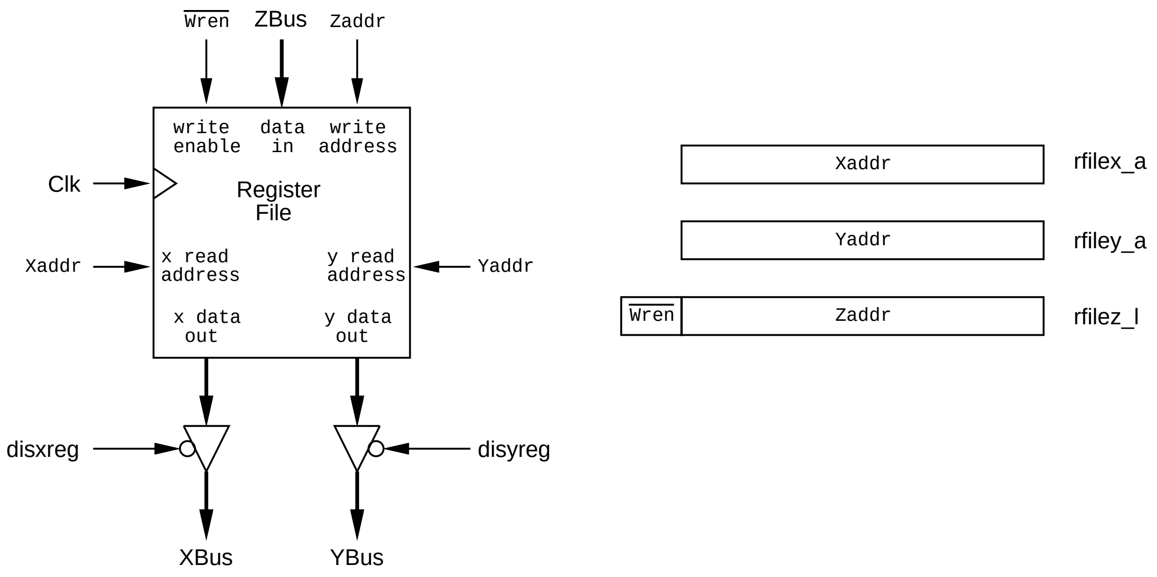


FIGURE 4-11: Floating point register file

The register file is implemented with two *Texas Instruments 74ACT8832A* parts, which are actually 32-bit registered ALUs. Since the ALU portion is not necessary, only the internal register files are used. The register file outputs are connected to the XBus and the YBus, while the register file input port is connected to the ZBus. The output to the XBus is enabled by the external signal disxreg, generated by the source memory subsystem, while the output to the YBus is enabled by the external signal disyreg, generated by the destination memory subsystem. The register file is clocked at 50Mhz by Clk, allowing a maximum bandwidth of 1.2 Gbytes/second.

The *TI 74ACT8832A* parts were used instead of the standard *Bipolar Integrated Technology, Inc. BIT 2210A* parts because they exhibit less latency. Additionally, they reduce the chip count by half, thus reducing board space, and dissipate 20 times less power.

The register file is controlled by three microinstructions, rfilex_a, rfiley_a, and rfilez_l. rfilex_a is an *align* control signal, which specifies:

`Xaddr` Register file address. The addressed register contents can be transferred to the XBus, depending on the value of the external line disxreg.

rfiley_a is an *align* control signal, which specifies:

`Yaddr` Register file address. The addressed register contents can be transferred to the YBus, depending on the value of the external line disyreg.

rfilez_l is a *lead* control signal, which specifies:

`Zaddr` Register file address. The contents of the register file input from the ZBus can be stored in the specified register in the rising edge of Clk if $\overline{\texttt{Wren}}$ is enabled.

$\overline{\texttt{Wren}}$ Register file write enable. Active low, allows data from the ZBus to be stored in the register specified by `Zaddr`.

Figure 4-12 illustrates the most relevant timing relationships for the floating point unit (FPU), the register file, and their interaction with the microprogrammed control. The μAddress register specifies the current microinstruction being executed. bitinst_l and rfilex_a are shown as examples of generic *lead* and *align* control signals. There are two basic constraints in the interaction between the register file and the floating point unit, and both must be smaller than the clock cycle period: the register file to XBus, YBus output data access time ($t_{XOD} \leq 16ns$) plus the setup time for the data input in the FPU ($t_{SX} \leq 3ns$), and the FPU to ZBus output data delay ($t_{ZOD} \leq 12ns$) plus the setup time for the register file data input ($t_{SZ} \leq 4ns$). In terms of the microprogrammed control and the FPU interaction, it is important that all *lead* signals issued by the control unit arrive to the FPU at least $t_{SX} \leq 3ns$ before the rising edge of the Clk. Since the setup time for the Tsel and Msel signals is substantially higher than the other FPU setup times, it was necessary to change these signals into *align* type signals, and advance the generation of these control signals in the microprogram by one cycle.

One system timing constraint that deserves special attention is the main control's branch decision logic input from the FPU. It is important that the sum of the FPU to flags output data delay ($t_{FOD} \leq 12ns$) plus the setup time for the microcontrol address register selection logic ($t_{S\mu} \leq 3ns$) be smaller than the clock period minus the small skew between Clk and μClk. This constraint is part of the critical path represented by the time required by the controller to issue an FPU compare instruction and execute or not a branch that depends on the instruction's

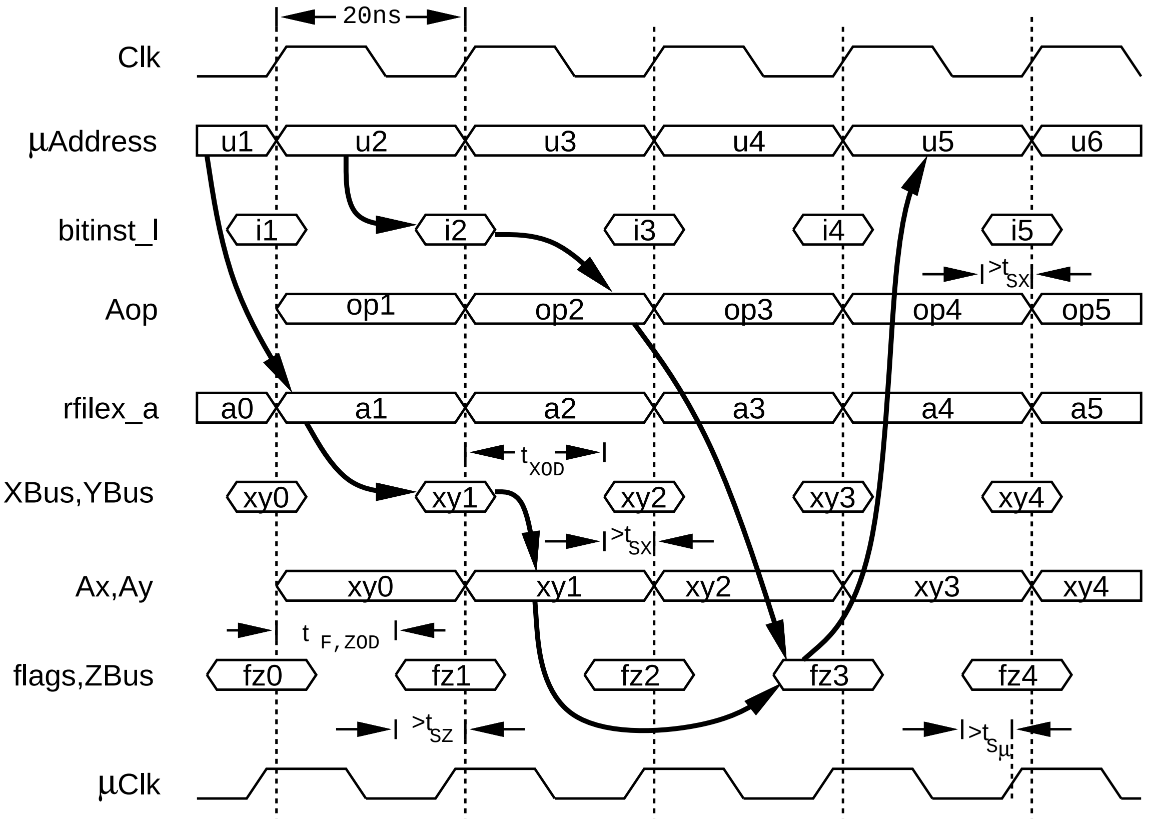


FIGURE 4-12: Floating point unit and register file timing

result. This situation is highlighted by the heavy arrows in Figure 4-12. Microinstruction address u2 specifies a floating point compare instruction i2. This instruction will be executed in the next clock cycle and its results will become available (fz3) in the FPU flag output pins after two clock cycles, thus permitting the execution of a conditional branch to the microinstruction address u5. An extra latency cycle is necessary if the compare instruction operands are kept in the register file, as highlighted by the heavy arrows in the path from the μAddress register to rfilex_a, then to the XBus, YBus, and then to the Ax, Ay registers, and finally to the flags.

All the timing constraints in this section can be satisfied if the suggested parts are used and the clock period is at least 20ns. For simplicity, the hold time constraints are not shown in Figure 4-12, but can also be met with the proposed clock cycle period.

Floating point units with speeds up to 200 Mhz have been reported in the literature [Dobberpuhl92], representing an immediate opportunity to quadruple the floating point subsystem speed. By the end of the decade, floating point units operating at 500 Mhz or beyond would be able to achieve a tenfold increase in the FPU speed.

The IC count for the floating point system is one 395-pin PGA package and two 208-pin PGA packages, with an estimated power dissipation of 28 Watts, which will require a large heat sink for qthe *BIT 2130* part and forced ventilation.

4.2.4 Memory System

The memory system consists of the three major components: the index memory, the source memory and the destination memory subsystem. In the following, each of these subsystems will be discussed separately.

Index Memory Subsystem

The index memory subsystem consists of the address generator, memory array, bidirectional transceivers between QBus and HBus, and the memory controller. Figure 4-13 depicts in detail the major components of the index memory subsystem and the associated microprogram control fields.

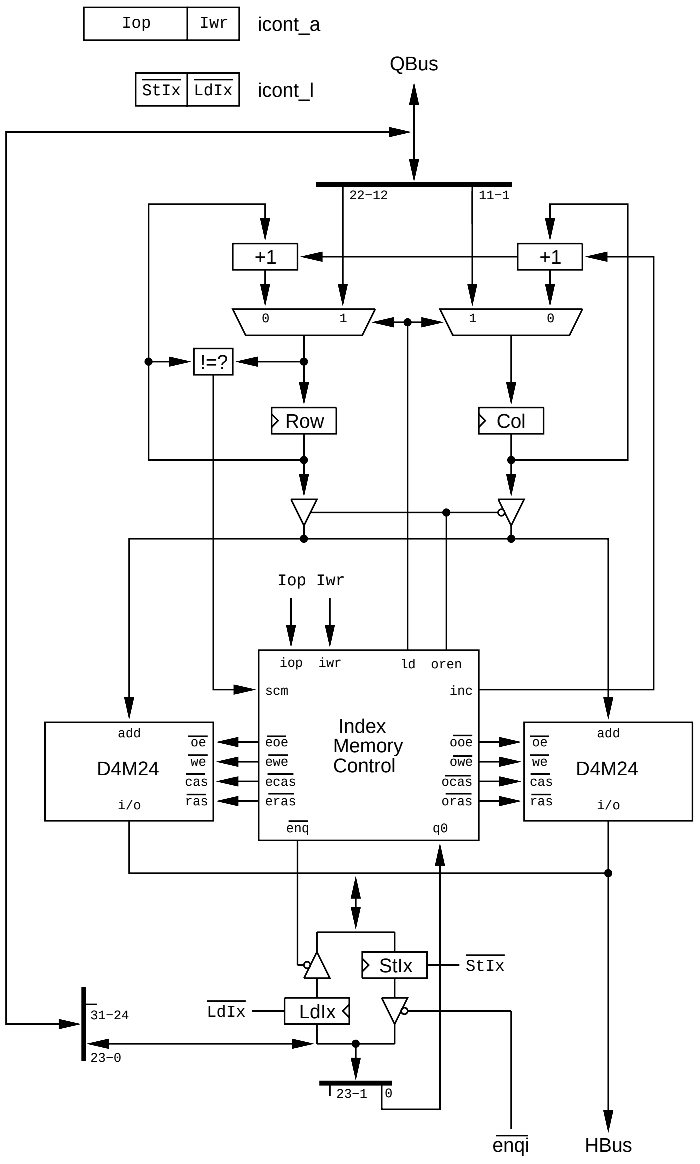


FIGURE 4-13: Index memory subsystem and its associated microinstruction fields

The address generator consists of two address registers, Row and Col, and their associated logic. Each register holds eleven address bits. The peripheral logic provides these registers parallel load from the QBus and Row-Col pair cascaded increment under control of the `ld` and `inc` signals. The associated logic also includes a comparator that checks if a memory page miss happened in the case the address being loaded in the Row register differs from the previous contents, or if there is a carry propagating from the Col register into the Row register during an increment operation. These registers and the peripheral logic can be implemented with three *Cypress PAL22V10C* devices. The tri-state outputs of the PAL devices can be directly connected to the memory array address lines, as they only have to drive a capacitance in the order of 60 pF. This structure reduces the memory access latency to two valid clock cycles, which is required in the implementation of the high-performance sparse matrix row update operation, as discussed in Section 4.2.2.

The two-way interleaved memory array consists of two groups of six 4 Mword by 4-bit common I/O DRAM devices, such as the *Texas Instruments TMS416400*. These twelve devices account for a total of 8M 24-bit words. All DRAM devices share the same address and $\overline{\texttt{ras}}$ control lines, while other DRAM control lines are driven independently for the even and odd memory banks.

The registered bidirectional transceivers between QBus and HBus can be implemented with three *Texas Instruments 74ABT2952* devices. The tri-state output enable for the LdIx register is provided by the memory controller, and is activated when a memory write cycle is executed. The tri-state output enable for the StIx register is an external signal $\overline{\text{enqi}}$, supplied by the QBus decode logic. The register load is independently enabled by the icont_l microinstruction field, as follows:

$\overline{\texttt{StIx}}$ Active low, enables loading data from the HBus into the StIx register on the rising edge of the Clk signal.

$\overline{\texttt{LdIx}}$ Active low, enables loading data from the QBus into the LdIx register on the rising edge of the Clk signal.

The memory controller consists of two tightly coupled synchronous finite state machines that take as input the icont_a microcontrol commands, the least significant bit of the QBus, and the `scm` flag indicating a memory page miss. It generates the appropriate control signals for the memory array, `inc` and `ld` signals for the address generator, a row/column address buffer selection signal (`oren`), a stall request signal stallreq, and a signal enabling LdIx register to drive the HBus during a memory write operation, $\overline{\text{enq}}$. The index memory control block can be implemented with three *Cypress PAL22V10C* devices.

The icont_a microinstruction specifies:

`Iwr` Used in conjunction with `Iop`, specifies the type of the access, in case of a valid memory cycle. When low, indicates an index memory write operation, and when high indicates an index memory read operation.

Iop Specifies the memory operation to be executed, along with the type of address to be used:

00	No memory operation (nop).
01	Valid access to current Row, Col address.
10	Valid access to the address to be loaded in the Row, Col from the QBus. (ld)
11	Valid access to the address obtained by incrementing the Row-Col pair. (inc)

Figure 4-14 illustrates a sequence of index memory cycles that highlights the most relevant aspects of the memory controller. Figure 4-14 provides a fuzzy idea of timing information, focusing on the sequencing aspects of the memory operation. Control instructions labeled inc2* and ld4* in Figure 4-14 cause a memory page miss.

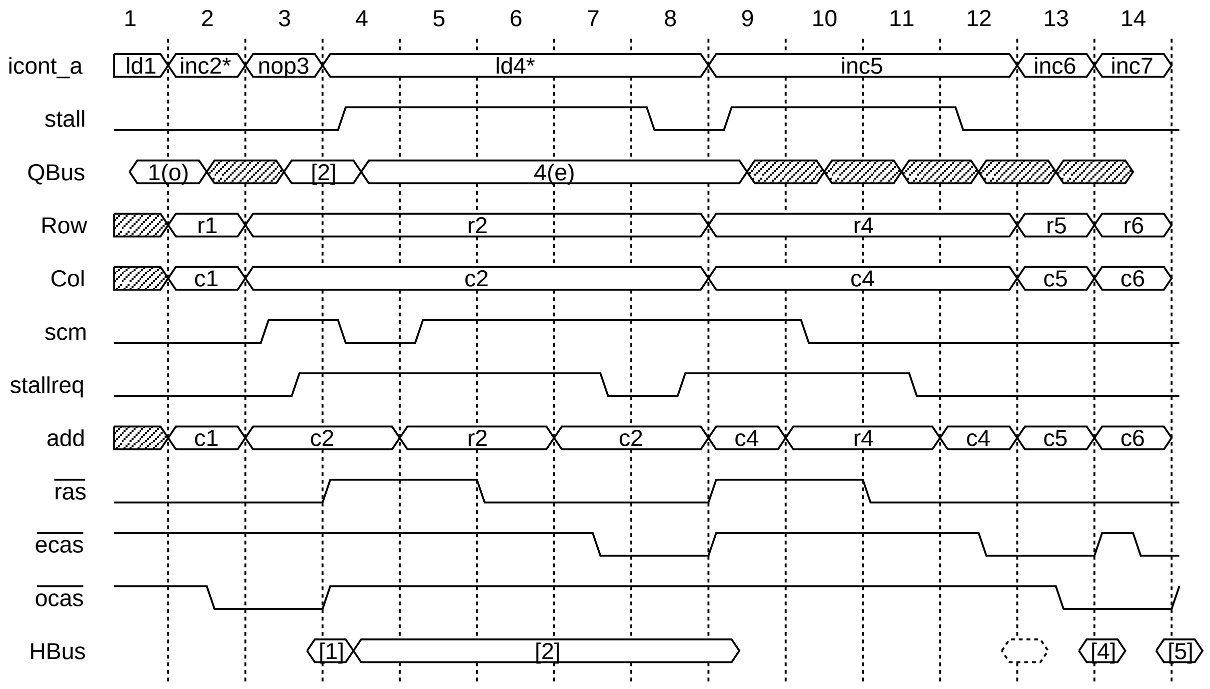


FIGURE 4-14: A relevant sequence of operations in the index memory subsystem

As mentioned earlier, the index memory latency is two *valid* cycles. For instance, the data [1], corresponding to the memory operation ld1 is available at the end of the clock cycle 3, two valid cycles after the memory access was requested. By the same token, data [2] is only written to the memory on the next valid cycle, 8, as cycles 4 through 7 correspond to processor stall cycles. During cycle 8, the data loaded in the LdIx register at the end of cycle 3, is written to memory. For write operations, data to be written in memory should be loaded in the LdIx register one valid clock cycle after the memory operation is requested.

The second memory request, inc2* causes the Col counter to overflow, requiring a memory access cycle to a different memory page. This situation is detected by the !=? block in Figure

4-13 and corresponds to the scm line driven high during cycle 3 in Figure 4-14. The index memory controller will detect the memory page miss and in response will issue a stallreq signal for four cycles, which in turn causes the main control to stall the processor during cycles 4 through 7. Since cycle 3 is valid, the memory controller considers the first operation (ld1) finished, and starts to service the second memory request. In order to access data in a different memory page, the $\overline{\text{ras}}$ line must be pulsed high for writing back the current memory page into the array and precharging the sense bitlines. The pulse duration is two clock cycles, and the falling edge of the $\overline{\text{ras}}$ signal in end of clock period 5 will strobe the row address r2 in the DRAM internal address registers. The inc2* operation is then completed at the end of clock cycle 8.

It is important to notice that the index memory controller must accept and store for future use whichever instruction is issued at the end of clock cycle 3, along with the relevant data. Since nop3 was requested, no action would be taken. However, if a ld3 was requested at the end of clock cycle 3, there would be no address register to hold the extra address. For this reason, the system software must make sure that a ld request can only follow a nop request, which is the only safe way of preventing data loss without adding extra hardware. Even though inc requests can cause memory page misses, they can be easily handled by keeping one extra flag in the controller that postpones the requested increment until the current operation is finished.

An interesting ability of the controller is the handling of nops and memory page misses, which can actually reduce the number of processor stall cycles. During clock cycle 8, the controller has knowledge that in order to honor the ld4* request it will be necessary to start a memory page miss cycle, and that the next valid cycle would be devoted to output dummy data, thus completing the nop3 request. Instead of completing the nop3 service, the index memory controller issues a stallreq immediately, causing cycles 9 through 11 to be processor stall cycles. The controller also proceeds with the memory page miss protocol as discussed before, but instead of requesting four stall cycles as usual, it will only request three stall cycles, as shown in Figure 4-14. The data present in HBus at the end of the cycle 12 is irrelevant, finishing the nop3 task service. The data corresponding to the ld4* request, [4], is available at the end of cycle 13.

The last two cycles in Figure 4-14 represent the most effective way of accessing index memory data. Typically, in the beginning of a sparse matrix row update operation, the initial address of the column indices for the source row would be loaded in the Row, Col registers, and increment operations would be requested for scanning all row indices in sequence. In this mode, there would be very few memory page misses, as statistically there is only one chance in 4096 of a memory page miss occurring during an increment operation, and data can be read from alternate memory banks in each clock cycle. Every clock cycle the contents of the Col register would be incremented and placed in the add address lines. The memory controller would then alternate issuing $\overline{\text{ecas}}$ and $\overline{\text{ocas}}$. The falling edge of each $\overline{\text{cas}}$ signal will strobe the column address in the DRAM internal address registers. The controller would also issue $\overline{\text{eoe}}$ and $\overline{\text{ooe}}$ in alternate cycles to read data through HBus at the maximum rate of one 24-bit word per clock cycle.

Figure 4-15 depicts in detail the most critical signal timing constraints that must be satisfied for the proper operation of the index memory system. The $\overline{\text{ras}}$ signal is assumed low throughout the time period shown, so only column addresses (address & 0x7fe) are present in the add address lines. The first and second memory cycles correspond to a write to the even memory bank, address b. During the second and third cycles there is a data read from the odd memory bank, address b+1.

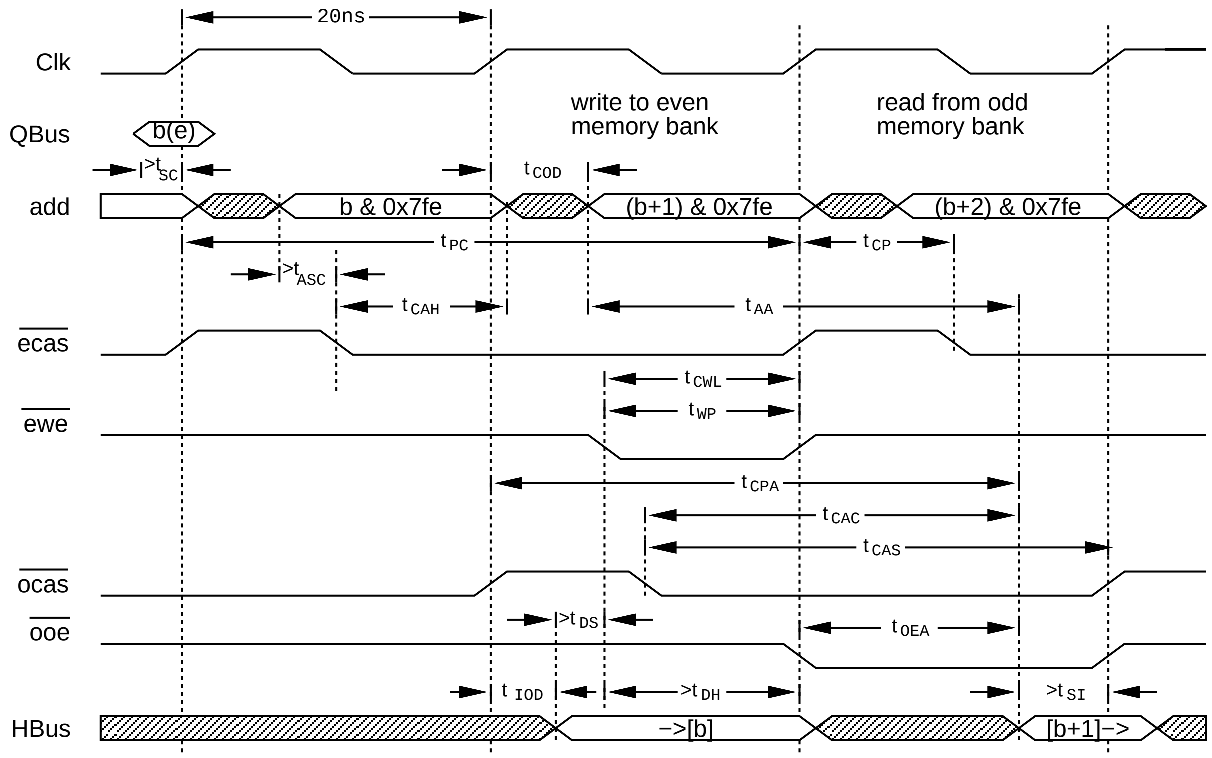


FIGURE 4-15: Index memory subsystem timing

Table 4-1 summarizes the index memory subsystem switching characteristics and timing requirements depicted in Figure 4-15 for the devices previously mentioned. Using the notation of Figure 4-15 and Table 4-1, and denoting the clock period as T_{Clk}, the following set of constraints must be satisfied for the proper operation of the index memory subsystem:

$$t_{PC} \leq 2 \times T_{Clk} \tag{4.1}$$

$$t_{COD} + t_{AA} + t_{SI} \leq 2 \times T_{Clk} \tag{4.2}$$

$$t_{CPA} + t_{SI} \leq 2 \times T_{Clk} \tag{4.3}$$

$$t_{COD} + t_{ASC} + t_{CAH} \leq T_{Clk} \tag{4.4}$$

Symbol	Description	Min(ns)	Max(ns)
t_{AA}	Access time from column address		30
t_{CAC}	Access time from $\overline{\mathsf{cas}}$ low		15
t_{CPA}	Access time from column precharge		35
t_{OEA}	Access time from $\overline{\mathsf{oe}}$ low		15
t_{COD}	Clock to Col register output delay		6
t_{IOD}	Clock to LdIx register output delay		5
t_{SC}	Col register input setup	3	
t_{ASC}	Column address setup before $\overline{\mathsf{cas}}$ low	0	
t_{CWL}	$\overline{\mathsf{oe}}$ low setup before $\overline{\mathsf{cas}}$ high	15	
t_{DS}	Data setup time	0	
t_{SI}	StIx register input setup	2	
t_{PC}	Page-mode read or write cycle time	40	
t_{CP}	Pulse duration, $\overline{\mathsf{cas}}$ high	10	
t_{CAS}	Pulse duration, $\overline{\mathsf{cas}}$ low	15	
t_{WP}	Write pulse duration	15	
t_{CAH}	Column address hold time after $\overline{\mathsf{cas}}$ low	10	
t_{DH}	Data hold time	10	

Table 4-1: Index memory subsystem switching characteristics and timing requirements

$$t_{OEA} + t_{SI} \leq T_{Clk} \tag{4.5}$$

$$t_{IOD} + t_{DS} + t_{CWL} \leq T_{Clk} \tag{4.6}$$

Constraints 4.1 through 4.3 refer to $\overline{\mathsf{cas}}$ precharge and access time constraints. Constraint 4.4 is linked with the setup and hold times of the address lines add in respect to the $\overline{\mathsf{cas}}$ signal. Constraint 4.5 is related with data read and output enable access. Finally, constraint 4.6 is related to the setup and hold times of the data to be written in respect to the write enable signal $\overline{\mathsf{we}}$. All conditions can be satisfied if $T_{Clk} \geq 20ns$.

Experimental synchronous and cached dynamic RAMs have been reported operating at 100 Mhz [Dosaka92], and promise to operate at frequencies up to 500 Mhz by the end of the decade. Using interleaving and other techniques described in this section, one can envision the feasibility of a tenfold increase in the index memory subsystem speed by the end of the decade.

The IC count for the index memory subsystem is nine 24-pin DIP packages accounting for the *PALs* and the registers, and twelve 28-pin plastic ZIP packages that compose the DRAM array. The estimated power dissipation is 12 Watts.

Source Memory Subsystem

The source memory subsystem consists of the address generator, memory array, the SrcZ register, two FIFOs, and the memory controller. Figure 4-16 depicts in detail the major components of the source memory subsystem and the associated microprogram control fields.

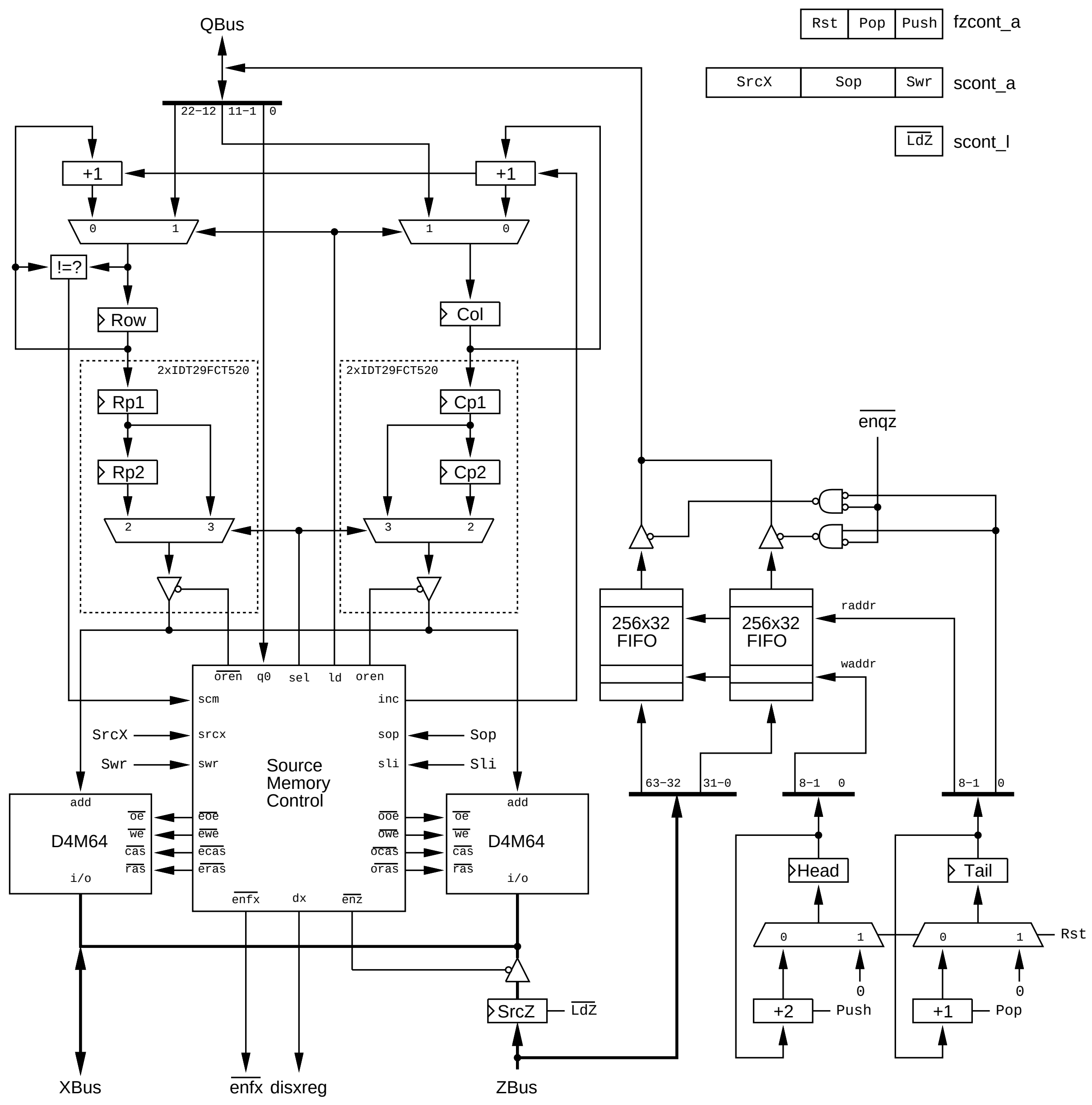


FIGURE 4-16: Source memory subsystem and its associated microinstruction fields

The address generator consists of the address registers Row, Col and their associated logic; and the pipeline buffer registers that drive the large DRAM array capacitance. The Row, Col register pair has the same capabilities as the equivalent set of registers in the index memory subsystem. The major difference between these sets of registers is the direct control by the Sop instruction field — as long as the current processor cycle is valid, the Row, Col registers will accept random data ld and inc requests without any sequencing restriction. This is possible because the pipeline registers can store pending requests and addresses in the case of a memory page miss. The Row, Col registers and associated logic can be implemented with three *Cypress PAL22V10C* devices. The DRAM array address lines add have a capacitive load of 160 pF, and must be driven by devices that can handle high output currents such as the *Integrated Device Technology, Inc. IDT29FCT520*. The source memory latency is three valid clock cycles, meeting the address driving requirements and still providing high speed in the implementation of the sparse matrix row update operation, as discussed in Section 4.2.2.

The two-way interleaved memory array consists of two groups of sixteen 4 Mword by 4-bit common I/O DRAM devices, such as the *Texas Instruments TMS416400*. These thirty-two devices account for a total of 8M 64-bit words. All DRAM devices share the same address and $\overline{\texttt{ras}}$ control lines, while other DRAM control lines are driven independently for the even and odd memory banks.

The registered buffer from the ZBus to the XBus can be implemented with eight *Texas Instruments 74ABT2952* devices. The tri-state output enable for the SrcZ register is provided by the memory controller, and is activated by the SrcX decode logic. The register load is independently enabled by the scont_l microinstruction control, as follows:

$\overline{\texttt{LdZ}}$ Active low, enables loading data from the ZBus into the SrcZ register on the rising edge of the Clk signal.

The FIFOs can be implemented with four *Texas Instruments 74ACT7803* devices, and its associated control can be implemented with one *Cypress PAL22V10C* device. The *PAL22V10C* holds only the the least significant bit of the Tail register and some output enable and read enable decode logic, while the remaining functionality is implemented within the four *74ACT803* devices. The tri-state output enable of these devices is activated by the $\overline{\text{enqz}}$ external signal, generated by the QBus decode logic. The FIFO operation is controlled by the fzcont_a microinstruction, specified as follows:

Push When high, pushes the contents of the ZBus into two consecutive FIFO memory positions, thus effectively incrementing the Head register by two.

Pop When high, pops one 32-bit word from the FIFO address specified by the Tail register, thus effectively incrementing it by one.

Rst When high, resets both FIFOs, thus flushing out all FIFO data and effectively loading zero on both Head and Tail registers.

In fact, when Push is active, 64-bit data from the ZBus is pushed simultaneously in all FIFO inputs, while the activation of Pop will only pop 32-bit data from the FIFO devices that are selected by the least significant bit of the Tail register, kept in the FIFO control logic. Typically, the FIFOs will be used to read in descriptor data from the source memory through the floating point unit in a single burst, acting as a cushion for the task descriptor access. With this mechanism, the memory page misses that would incur by switching between source data access and descriptor access before each row update operation is eliminated, thus substantially improving the system performance. In spite of its pivotal importance, the added cost of the FIFOs in terms of board space and power dissipation is minimal: a registered buffer from the source memory or floating point unit to the QBus would be required in any case, and the board area taken either by FIFOs or by common registers with tri-state output buffers is comparable.

The source memory controller consists of pipeline registers that store pending values of the least significant bit of QBus, and the scont_a microinstruction, feeding these values directly into the inputs of two tightly coupled finite state machines (FSMs). The FSMs generate the appropriate control signals for the memory array; inc, ld, and sel signals for the address generator; and a pair of row/column pipelined address buffer select lines (oren and $\overline{\mathtt{oren}}$). Some additional decode logic linked with the FSMs generates the $\overline{\mathtt{enz}}$ signal that enables the contents of the SrcZ register to drive the XBus; the disxreg signal that disables the floating point register file tri-state drivers to the XBus, as discussed in Section 4.2.3; and the $\overline{\mathsf{enfx}}$ that enables the FIFO output drivers to the XBus in the external bus interface, which will be discussed in detail in Section 4.2.5. The source memory control can be implemented with four *Cypress PAL22V10C* devices and a *Texas Instruments 74ABT2952* device that provides enough output current for driving capacitances up to 200 pF in the memory array control lines.

The scont_a microinstruction specifies:

Swr Used in conjunction with Sop, specifies the type of the access, in case of a valid memory cycle. When low, indicates a source memory write operation, and when high indicates a source memory read operation.

Sop Specifies the memory operation to be executed, along with the type of address to be used:

00	No memory operation (nop).
01	Valid access to current Row, Col address.
10	Valid access to the address to be loaded in the Row, Col from the QBus. (ld)
11	Valid access to the address obtained by incrementing the Row-Col pair. (inc)

SrcX Specifies which device will be driving the XBus in the valid cycle that the instruction specified by Sop is executed, according to the following table:

00	XBus is driven by the DRAM array.
01	XBus is driven by the SrcZ register.
10	XBus is driven by the floating point register file.
11	XBus is driven by the external bus interface FIFO.

Figure 4-17 depicts a sequence of source memory cycles that highlights the most relevant aspects of the memory controller. Like Figure 4-14, it provides a fuzzy representation of the timing information, focusing on the aspects of sequencing. In order to highlight the differences with the index memory system, the same sequence of addresses is presented to the controller. inc2* in Figure 4-17 corresponds to a read operation, in order to exemplify extended stall cycles caused by other units.

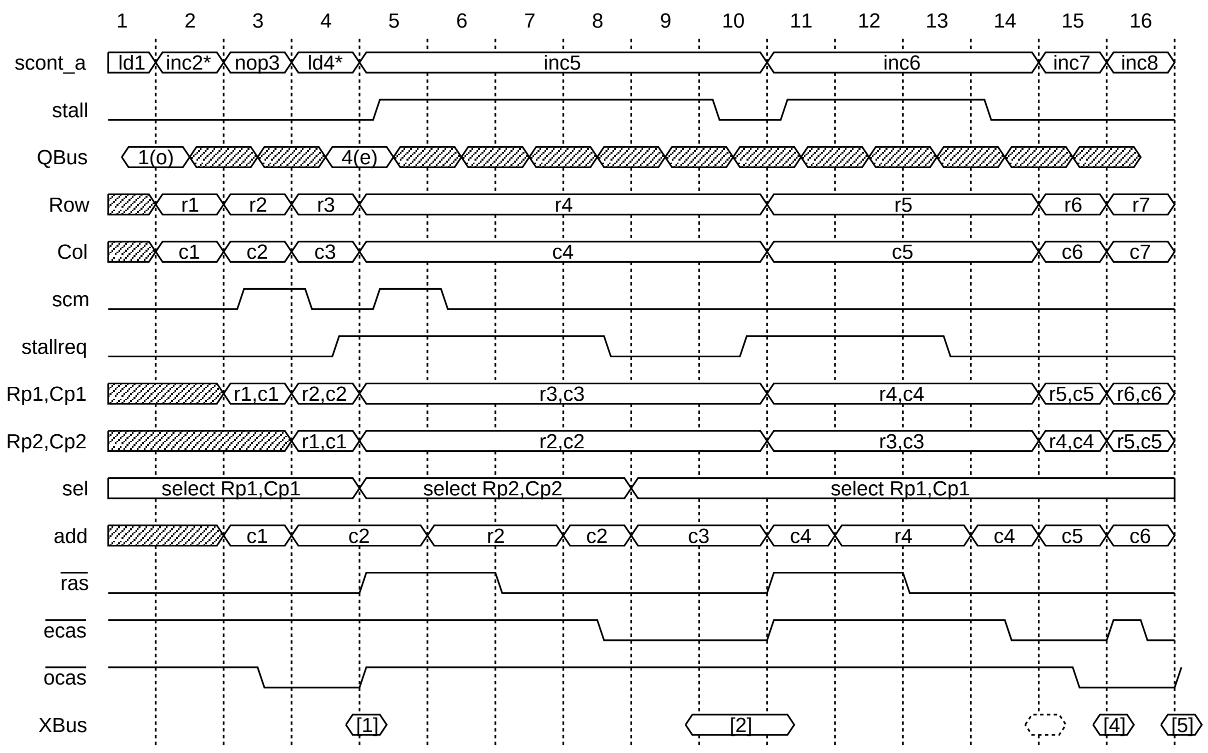


FIGURE 4-17: A relevant sequence of operations in the source memory subsystem

In contrast to the index memory, the source memory latency is three *valid* cycles, so the data [1], corresponding to ld1 is available in the end of the clock cycle 4. This extra delay requires additional address registers for holding addresses for pending operations.

The sel signal will be selecting the contents of Rp1,Cp1 during normal memory operation. However, in the case of a memory page miss, sel normally selects the contents of Rp2,Cp2, as is the case in cycles 5 through 8 in Figure 4-17. During cycles 11 through 14 the controller selects the contents of Rp1,Cp1 to drive the address lines because the contents of Rp2,Cp2 do not correspond to a valid operation.

It is important to notice that the address generated by whichever operation was requested on cycle 3 is stored in the Rp1,Cp1 register pair, and the address generated on cycle 4 is kept in the Row, Col register pair. Even in the extreme case both these instructions were lds and caused memory page misses, the memory system would still be able to properly handle the requests by slowing down the processor and handling each operation separately.

Clock cycle 9 is a processor stall cycle, but the stall was not caused by the source memory system, as data [2] is available at the end of the cycle. In these cases, called extended stall cycles, the memory controller monitors the status of the stall line and will keep either or both $\overline{\text{cas}}$ signals in low state, depending on state of the access. Also, the $\overline{\text{oe}}$ signal will be asserted selecting the appropriate memory bank. The extended stall will continue until a valid processor cycle is executed, in which case the controller will consider the current task finished and will proceed to the next task. The index memory controller is also able handle these extended stall cycles.

The timing constraints that must be satisfied for the proper operation of the source memory system are the almost the same as the index memory system constraints. Referring to Figure 4-15, the differences between these systems are the additional latency cycle between the address load from the QBus and its output in the add address lines, and slight setup times t_{SI} and t_{SX} discrepancies.

Like the index memory subsystem, the source memory can also benefit from synchronous and cached dynamic RAMs operating at higher frequencies, and it is also possible to obtain a tenfold speed increase by the end of the decade.

The IC count for the source memory subsystem is twenty 24-pin DIP packages accounting for the *PALs* and registers, and the thirty-two 28-pin ZIP packages that compose the DRAM array. The estimated power dissipation is 28 Watts.

Destination Memory Subsystem

The destination memory subsystem consists of the ALU-based address generator, memory array, and the controller. Figure 4-18 depicts in detail the major components of the destination memory subsystem and its associated microprogram control fields.

The address generator is based on the *Integrated Device Technology, Inc. IDT7383L20* 16-bit ALU, handling address bits 1 through 16. Since the destination memory is two-way interleaved, special logic and registers with functionality similar to the ALU are provided for address bit 0, and could be implemented with one *Cypress PAL22V10C* device. The ALU contains two input registers Al and Af, and one output register Af that can be independently loaded under the microprogram control field dcont_l, as follows:

$\overline{\text{Enal}}$ Active low, enables loading data from the QBus into the Al input register on the rising edge of the Clk signal.

$\overline{\text{Enah}}$ Active low, enables loading data from the HBus into the Ah input register on the rising edge of the Clk signal.

$\overline{\text{Enaf}}$ Active low, enables loading data from the ALU into the Af output register on the rising edge of the Clk signal.

The ALU is a combinational block that under control of the six-bit Aop microinstruction field of dcont_a can perform a variety of simple 16-bit integer arithmetic and logic operations such as pass, add, subtract, bitwise and, bitwise or, etc. on its input operands, which can be any Al, Ah, or Af registers. The *Integrated Device Technology, Inc. High Performance Logic Databook* [IDT92] contains a detailed description of all ALU operations.

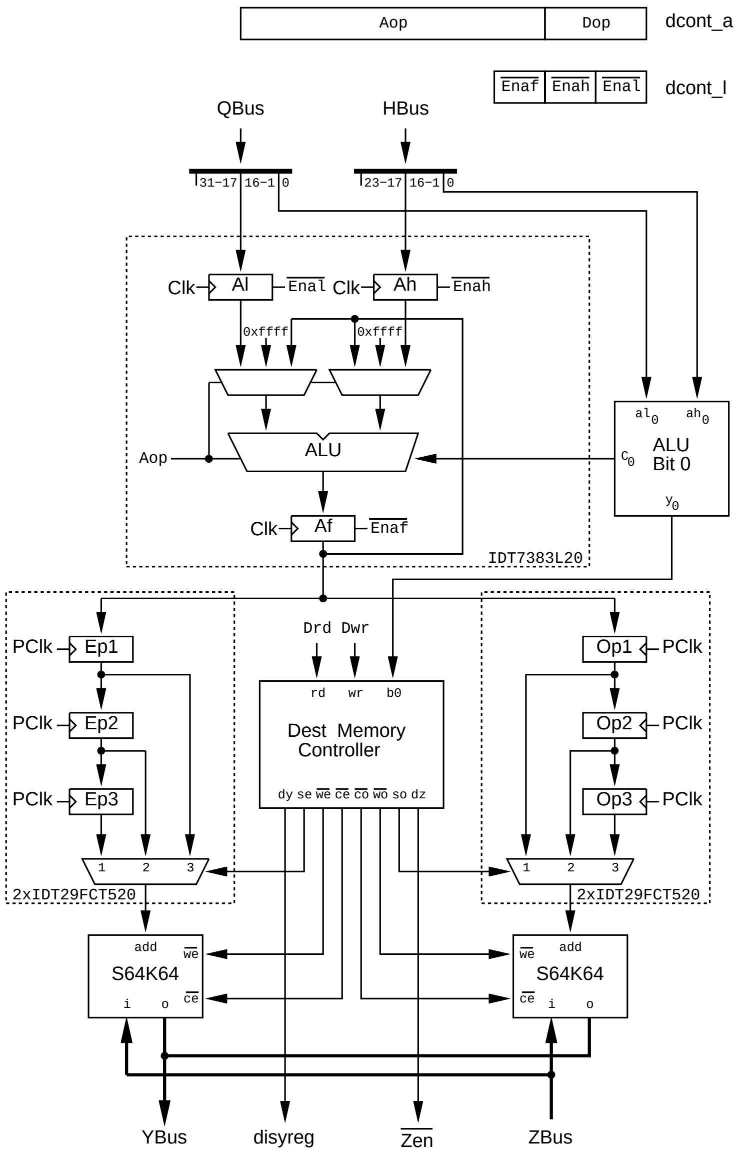


FIGURE 4-18: Destination memory subsystem and its associated microinstruction fields

Pipelined registers with buffered output such as the *Integrated Device Technology, Inc. IDT29FCT520* are necessary to hold writeback addresses and provide enough output drive current for the 160 pF capacitive load of each memory array. A total of four devices are used. A skewed version of the main processor clock, PClk, clocks the pipelined registers in order to align the SRAM array address transitions with the Clk signal, using the technique discussed in Section 4.1.1. A detailed description of the destination memory subsystem timing is depicted in Figure 4-20.

The two-way interleaved memory array consists of two groups of sixteen 64 Kwords by 4-bit separate I/O SRAM devices, such as the *Cypress CY7C192* devices. These thirty-two devices account for a total of 128K 64-bit words. The memory input pins of both data banks are directly connected to the Zbus, while the output pins are directly connected to the YBus in order to reduce the overall system latency. Separate address and control lines are provided for each memory bank.

The destination memory controller consists of pipeline registers that store pending values of the least significant bit of the address generated and the Dop microinstruction field, feeding these values directly into the inputs of a simple finite state machine (FSM). The FSM generates the appropriate control signals for the memory array; select signals for the pipelined address registers; the disyreg signal that disables the floating point register file tri-state output drivers to the YBus, as discussed in Section 4.2.3; and the $\overline{\text{Zen}}$ signal, which controls the Z floating point unit output register load, also discussed in Section 4.2.3. The destination memory controller can be implemented with three *Cypress PAL22V10C* devices and a *Texas Instruments 74ABT2952* device the provides enough output current for driving capacitances up to 160pF in the memory array control lines. The Dop field of the dcont_a microinstruction specifies the memory operation to be executed, according to the following table:

00	No operation (nop)
01	Memory read (r)
10	Memory write (w)
11	Memory read and result writeback in precisely two clock cycles. (rw)

In any clock cycle that neither memory bank is being read, the YBus will be driven by the floating point register file, as the destination memory controller issues a low output voltage in the disyreg signal.

Figure 4-19 depicts a sequence of destination memory cycles that highlights the most relevant aspects of the memory controller and ALU operations, using as an example the events related to a sparse matrix row update operation. Like 4-14 and 4-19, it focuses on the aspects of sequencing rather than representing precise timing information. The dcont_a depicted includes information on the Dop,Aop control pair. p specifies an ALU pass_Al operation, while + specifies an ALU add_Al_Ah operation.

On the beginning of the first cycle, the base address b, or the multiplier a_{ik} of the sparse matrix update, as described in Section 4.2.2, is loaded in the Al register from the QBus, and a ALU pass_Al operation is executed. The label b(e) on the QBus data means that b in this example is an address in the even memory bank. Only a read operation is requested for this address.

All other ALU operations depicted are sums of b and several indices i1, i2, etc. read from the HBus and used to compute the addresses of the data pertaining to the target row, or a_{ij} = [b + i], as described in Section 4.2.2. A memory read with corresponding writeback operation (rw) will be specified for all addresses computed in this fashion. Likewise QBus, data in the HBus is also labeled with either (e) or (o) to indicate an even or odd column index. In this example, the final address computed and stored in Af, will have the same even/odd pattern as the incoming

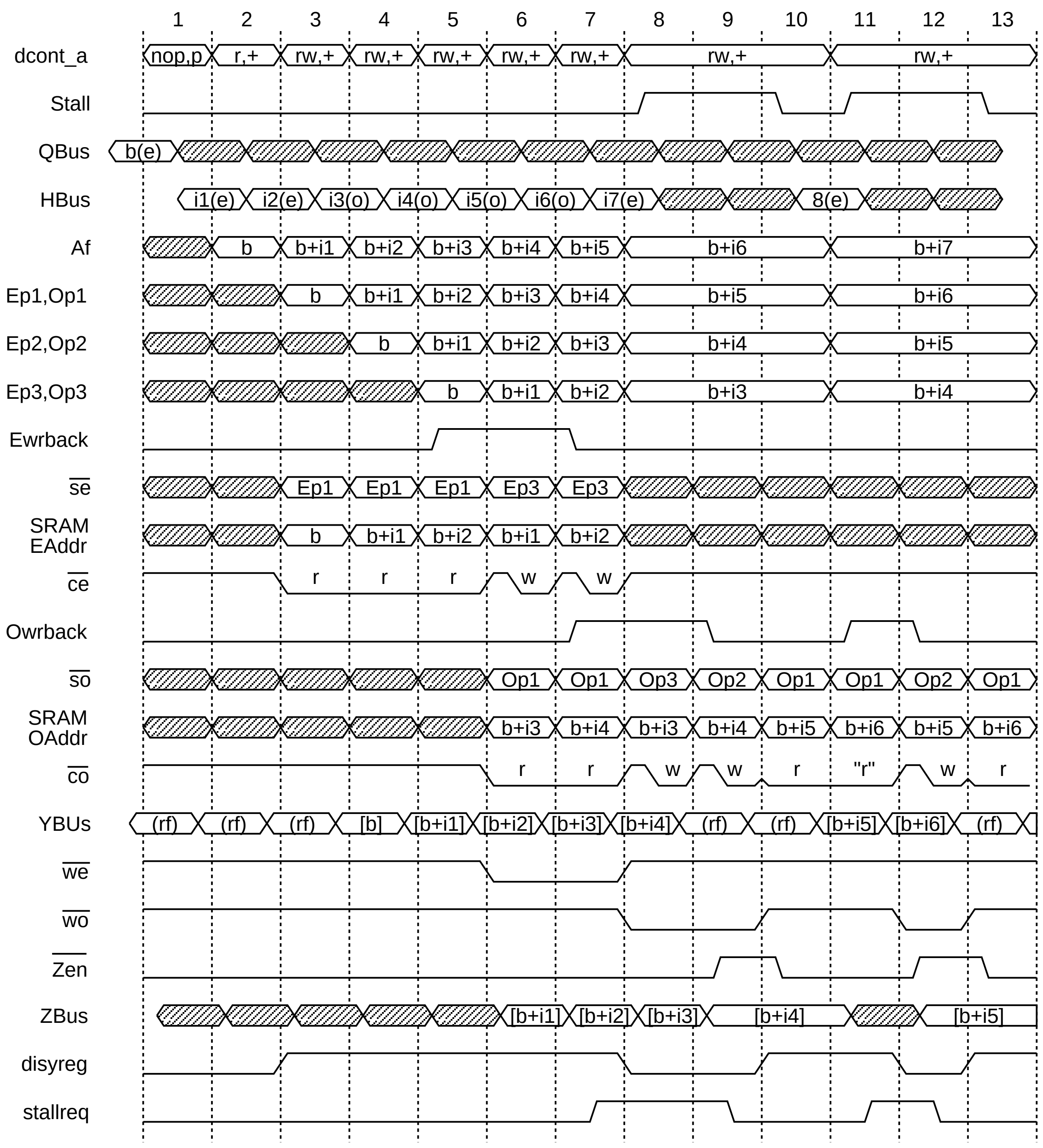


FIGURE 4-19: A relevant sequence of operations in the destination memory subsystem

column indices, as the base address was an even number.

During cycle 3, as well as in all valid clock cycles that do not correspond to a writeback operation, Ep1 and Op1 will be selected for the SRAM address lines. For normal operation, when a writeback is scheduled, either Ep3 or Op3 will be selected, depending on the memory bank that is the target of the writeback. For instance, cycles 6 and 7 correspond to writeback cycles in the even memory bank, and consequently Ep3 is selected, repeating b+i1 and b+i2 as even memory addresses. Correspondingly, the results of the *gaxpy* operations ([b+i1]= [b+i1] - [b] $\times$ s1) and ([b+i2]= [b+i2] - [b] $\times$ s2) will be present in the ZBus for writeback, and the $\overline{\text{we}}$ control line will be active. During these same clock cycles, new data is being read from the odd memory bank, thus achieving the maximum destination memory bandwidth. Usually, it is necessary to reorder the access to even and odd column indices in a sparse matrix in order to achieve the maximum destination memory bandwidth. A detailed discussion of how this is this reordering is implemented, and its impact on the overall performance of the system was presented in Section 2.2.1.

Sometimes it is not possible to reorder all column indices in order to achieve the maximum destination memory bandwidth. For instance, during clock cycle 8 in Figure 4-19, the data read for [b+i5] conflicts with the writeback for [b+i3], and during cycle 9 it conflicts with the writeback for [b+i4]. Since the writebacks cannot be delayed, they will be executed and these cycles will correspond to processor stall cycles, delaying the data read for clock cycle 10. It is also important to note that Op2 is selected during clock cycle 9 to accommodate for the stalled address pipeline during clock cycle 8.

Clock cycle 11 is a processor stall cycle, but the stall is not caused by the destination memory system, as data [b+i6] is available in the end of the cycle. Using the same strategy of source and index memory controllers in respect to extended stall cycles, the memory controller will monitor the stall line and will keep repeating the same read or write cycle until a valid cycle happens. However, clock cycle 12 corresponds to a writeback operation that cannot be delayed, so the extended stall cycle is aborted, the destination memory controller *forces* cycle 12 to be a processor stall cycle, the writeback is performed, and the system returns to the extended stall state during clock cycle 13, but it is broken immediately as it corresponds to a valid clock cycle.

Another important issue that must be dealt by the system software, otherwise expensive hardware would be required is the *operand bypass detection*. Considering the sequence of events on Figure 4-19, let us assume that in a generalized task with writeback operations, the situation i2= i1 could happen, so that data read from [b+i2] at the end of clock cycle 5 was actually supposed to be the data resulting from the writeback operation that started a cycle earlier, which is only going to be written back to memory during clock cycle 6. This situation can actually occur if the last *gaxpy* operand a_{ij} of one sparse matrix row update is the same element as the multiplier a_{ik} of the next row update. The operation could actually be correctly performed, assuming that the data present at the end of cycle 5 in the YBus was ignored, and the contents of the floating point unit ALU output were fed back to the input register My instead. This case can be predicted in the precompilation phase and the correct *bypass* scheduled in advance for sparse matrix factorization. Otherwise, extra hardware would be needed for detecting the address collision and taking the appropriate action.

Figure 4-20 depicts the most critical signal constraints that must be satisfied for the proper operation of the destination memory system. During the first cycle, a memory read from address d1 is performed, while in the second cycle, a memory write to address d2 is performed.

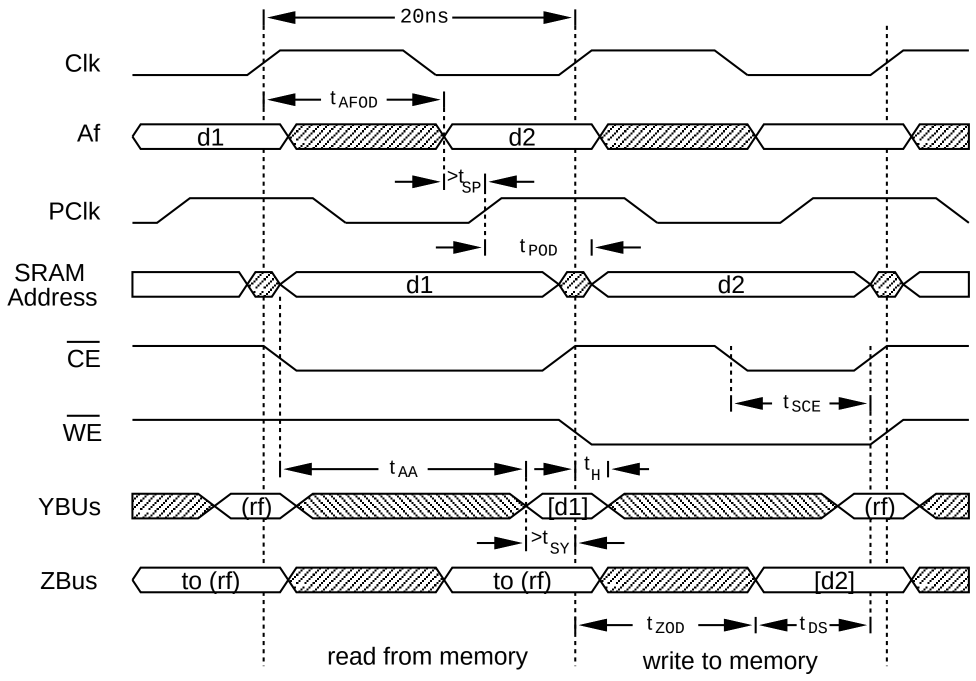


FIGURE 4-20: Destination memory subsystem timing

Table 4-2 summarizes the destination memory subsystem switching characteristics and timing requirements depicted in Figure 4-20, for the devices previously mentioned. Following the notation of Figure 4-20 and on Table 4-2, and denoting the clock period as T_{Clk}, the following set of constraints must be satisfied for the proper operation of the destination memory subsystem:

$$t_{AFOD} + t_{SP} + t_{POD} + t_{AA} + t_{SY} \leq 2 \times T_{Clk} \qquad (4.7)$$

$$t_{ZOD} + t_{DS} \leq T_{Clk} \qquad (4.8)$$

Constraint 4.7 is related to the read latency, from the moment the desired address is latched in the ALU output Af register until the output data is latched in the FPU unit input registers via the YBus. The skew between PClk and Clk must be tuned carefully to match the setup time of the pipeline register. Constraint 4.8 is related to the write cycle, representing the setup time constraints of the data to be written.

All the timing constraints in this section can be satisfied if the suggested parts are used and the clock period is at least 20ns. For simplicity, the hold time constraints are not shown in Figure 4-20, but can also be met with the proposed clock cycle period.

Experimental pipelined static RAMs have been reported operating at 500 Mhz [Chappell91]. Using interleaving and other techniques described in this section, it is possible to attain a tenfold speed increase in the destination memory subsystem.

Symbol	Description	Min(ns)	Max(ns)
t_{AA}	Address to valid data		15
t_{AFOD}	Clock to Af register output delay		11
t_{ZOD}	Clock to Z register output delay		12
t_{POD}	PClk to add output delay		7
t_{SP}	Pipeline register setup time	2	
t_{SY}	FPU register setup time	3	
t_{DS}	Data setup to write end	8	
t_{SCE}	$\overline{\text{CE}}$ low to write end	10	

Table 4-2: Destination memory subsystem switching characteristics and timing requirements

The IC count for the destination memory subsystem is one 68-pin PGA package containing the ALU, nine 24-pin DIP packages accounting for the *PALs* and registers, and the thirty-two 28-pin DIP packages that compose the SRAM array. The estimated power dissipation is 35 Watts.

4.2.5 Network Interface

The bus interface subsystem consists of four FIFOs buffering the flow of integer and double precision floating point data to and from IBus and DBus, registered bus transceivers that comply with the electrical bus interface protocol discussed in Section 4.1.2, and the control logic. Figure 4-21 depicts the overall organization of the bus interface.

The IOUT buffer is a 512-word by 33 bit FIFO that receives integer data from the QBus under microprogram command and sends it to the IBus under control of the transmit logic. Thirty-two bits from the QBus are used for data, and an extra bit from the microprogram control word is used for the generation of the Done signal during transmission. The IIN buffer is a 512-word by 33 bit FIFO that selectively receives data from the IBus under control of the receive logic. Thirty-two bit integer data can be read into the QBus under microprogram command, while the data corresponding to the incoming Done signal is the $\overline{\text{do}}$ flag, sent to the main control unit. The two FIFOs can be implemented with four *Texas Instruments 74ACT803* devices.

The DOUT buffer is a 512-word by 64 bit FIFO that receives double precision floating point data from the XBus under microprogram command and sends it to the DBus under control of the transmit logic. The DIN buffer is a 512-word by 64 bit FIFO that selectively receives data from the DBus under receive logic control. The data can be read into the XBus under microprogram control. These FIFOs can be implemented with eight *Texas Instruments 74ACT803* devices.

The registered bus transceivers depicted in Figure 4-21 latch signals coming to and from the interface network and translate between CMOS and BTL voltage levels, as discussed in Section

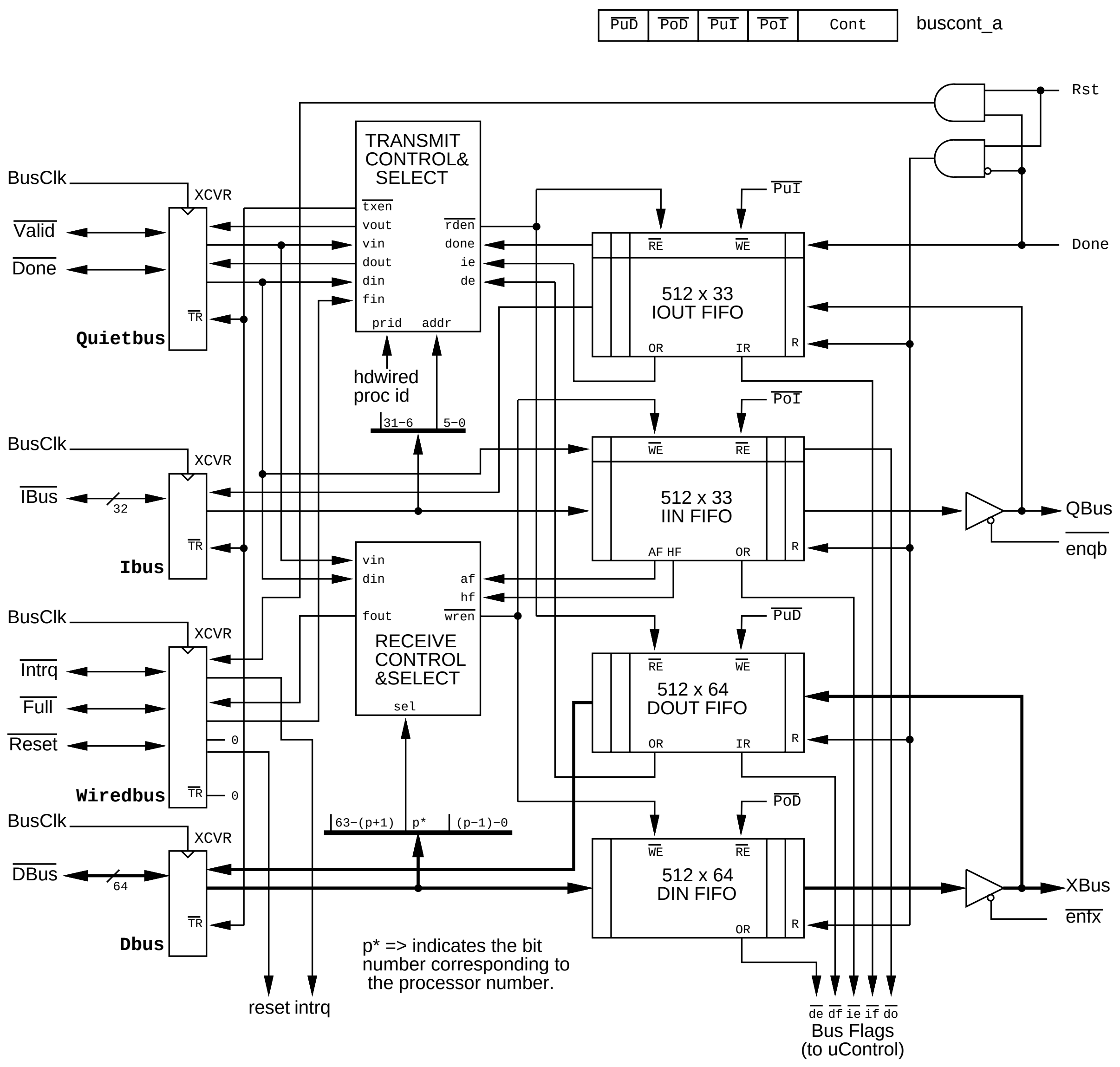


FIGURE 4-21: Interconnection interface and its associated microinstruction fields

4.1.2. Fourteen *Texas Instruments 74FB2033* devices are required for driving all the data and control signals.

The control section is divided into the transmit and receive control subsections. The transmit controller monitors the status of the Valid and Done input signals. Exactly one cycle after both signals become active, the *message control vector* is placed on the bus, as discussed in Section 4.1.2. The controller will then compare the six least significant bits of the IBus input with the hardwired process identification. If there is a match, the controller will then wait for the next occurrence of the Done and Valid signals, and a cycle later it will become the bus master. As soon as a processor gains bus ownership, the controller will start popping data from the IOUT and DOUT FIFOs, while watching the status of both ie and de signals to control the Valid signal output. Upon detecting the done signal from the IOUT FIFO, it will make sure that the output FIFOs are not empty, and then assert the vout signal along with dout. In the next cycle the *message control vector* will be transmitted, and the processor will relinquish bus control. If either IOUT or DOUT FIFO becomes empty when the done signal from IOUT is detected, the controller will not assert the Valid signal, as it is necessary to send the message control vector on the bus *exactly* one cycle after both Valid and Done signals are asserted. The transmit control logic is relatively simple and can be implemented with a single *Cypress PAL22V10C* device.

The receive controller also monitors continuously the status of the Valid and Done bus signals. Exactly one clock after both signals become active, the receiver controller fo the p-th processor will examine the p-th bit of the DBus to determine if the specific processor will be enabled to receive the next message data. If not, the processor will ignore all incoming data. If the processor is enabled, the receiver controller will push incoming data from the bus on both IIN and DIN FIFOs provided the vin signal is active. The receive control logic is very simple and can be implemented with a single *Cypress PAL22V10C* device.

The buscont_a microprogram instruction specifies:

Cont Contains the Rst (lsb) and the Done (msb) microcontrol signals. These two bits are used together to specify:

00	No operation (nop)
01	Resets all bus controllers and flushes all FIFO data (Rst)
10	Assert Done signal, flagging the last data in a message (Done)
11	Resets bus controllers, flushes all FIFO data and asserts the Intrq signal.

$\overline{\text{PoI}}$ Active low, pops one 33-bit word from the IIN buffer so that it could be read during the subsequent cycles to the QBus and the $\overline{\text{do}}$ flag.

$\overline{\text{PuI}}$ Active low, pushes into the IOUT buffer one 33-bit word formed by the juxtaposition of 32-bit data from the QBus and the Done microcontrol signal in buscont_a.

$\overline{\text{PoD}}$ Active low, pops one 64-bit word from the DIN buffer so that it could be read in the subsequent cycles to the XBus.

$\overline{\text{PuD}}$ Active low, pushes one 64-bit word from the XBus into the DOUT buffer.

It is important to emphasize that the system software is responsible for pushing into the IOUT, DOUT FIFO pair the correct message control vector after the last piece of data in the message, along with the Done control asserted, is pushed. Even though the hardware is designed in such way that bus conflicts will never happen, it is very likely that an irrecoverable error condition will be established if the the wrong message control vector is sent to the bus.

Figure 4-22 depicts in detail the most critical signal timing constraints that must be satisfied for the proper operation of the bus interface subsystem.

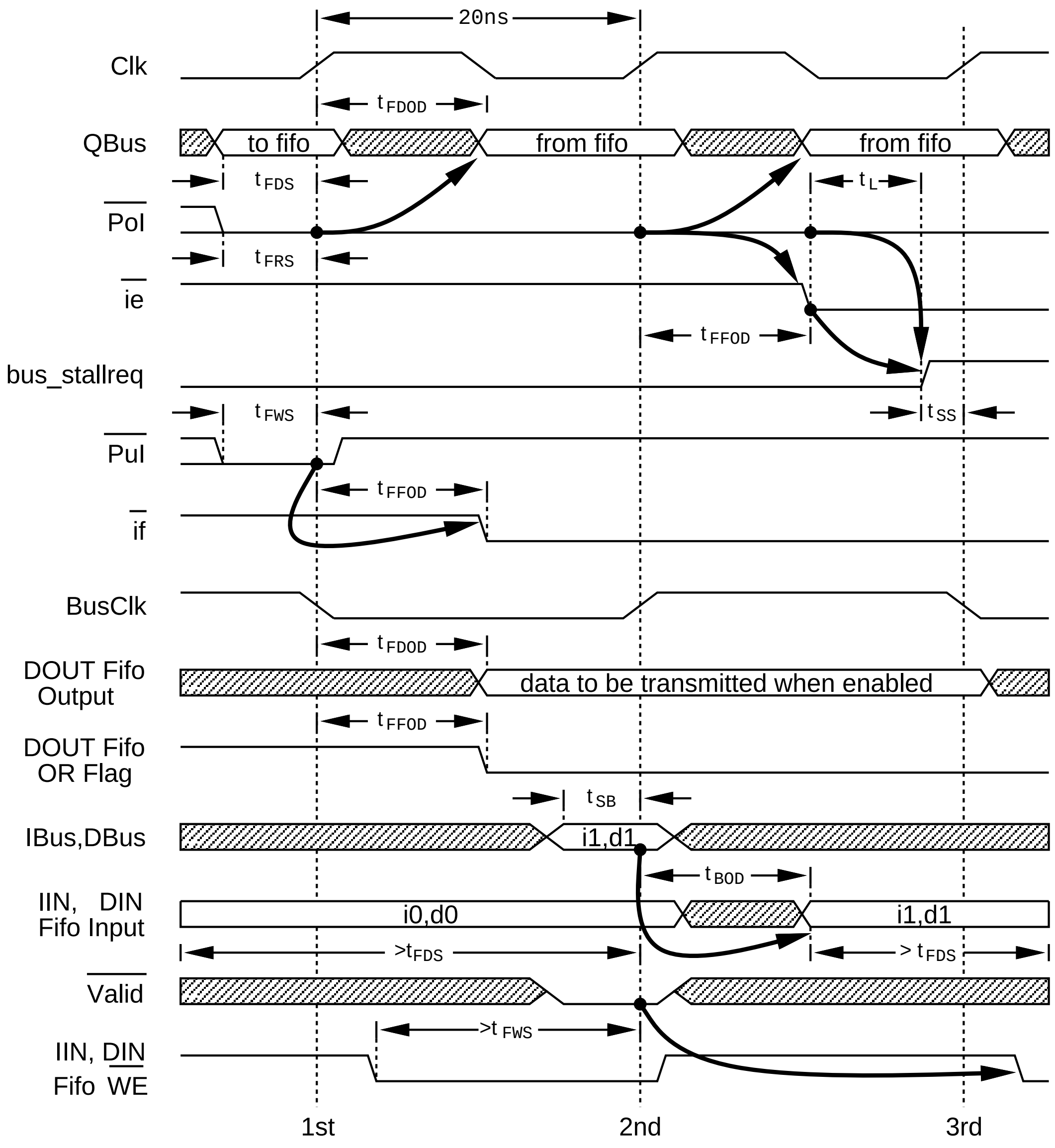


FIGURE 4-22: Network interface system timing

Let us first consider the interactions between the processor and the bus interface. In the first rising edge of the Clk signal data from the QBus is pushed into the IOUT Fifo under the command of the $\overline{\text{PuI}}$ microinstruction. This data push causes the IOUT FIFO to become almost full, as indicated by the assertion of the $\overline{\text{if}}$ flag. Any attempt to write extra data into this FIFO will cause the processor to stall. Also, data is successfully popped out of the IIN FIFO and read to the QBus during the first and second clock cycles, emptying it, as indicated by the assertion of the $\overline{\text{ie}}$ flag. The attempt to pop data from the empty FIFO, to be read during the third cycle causes the processor to stall, indicated by the assertion of the bus_stallreq signal.

Considering the interactions of the external bus and the interface, the first piece of data to be transmitted becomes available a few clock cycles after the first data was pushed into the

Symbol	Description	Min(ns)	Max(ns)
t_{FFOD}	Fifo clock to valid output flags		10
t_{FDOD}	Fifo clock to valid output data		15
t_{BOD}	Bus transceiver output delay		8
t_L	PAL logic propagation delay		7
t_{SB}	Bus input setup time	5	
t_{SS}	Stall request setup time	3	
t_{FWS}	Fifo write enable setup time	5	
t_{FRS}	Fifo read enable setup time	5	
t_{FDS}	Fifo data input setup time	5	

Table 4-3: Network interface system switching characteristics and timing requirements

DOUT FIFO from the XBus. Since the IOUT FIFO is full, the first piece of data from the QBus must be long waiting to be transmitted. The valid data will be transmitted as soon as the processor acquires bus ownership. Assuming that the data read from the bus is valid, and the processor is enabled to receive the contents of the current bus message, the data pair $\{\mathsf{i0}, \mathsf{d0}\}$ is pushed in the IIN and DIN FIFOs at the second rising edge of the Clk signal. Likewise, valid data $\{\mathsf{i1}, \mathsf{d1}\}$ was available in the bus and will be pushed into the FIFOs at the fourth rising edge of the Clk signal.

Table 4-3 summarizes the network interface system switching characteristics and timing requirements depicted in Figure 4-22 for the devices previously mentioned. Assuming that T_{Clk} is the clock period and using the notation of Table 4-3 and Figure 4-22, the most difficult constraint to satisfy is:

$$t_{FFOD} + t_L + t_{SS} \leq T_{Clk} \tag{4.9}$$

Constraint 4.9 can be satisfied if $T_{Clk} \geq 20ns$. All other constraints are easily satisfied.

The IC count for the network interface system is twelve 56-pin SSOP packages accounting for the FIFOs, fourteen 48-pin SSOP packages for the BTL transceivers, and two 24-pin DIP packages accounting for the *PALs*. The estimated power dissipation is 18 Watts.

4.2.6 Microprogrammed Controller

The microprogrammed controller consists of the microprogram program counter PC and associated branch logic, the microprogram memory, the Nop and Align registers, the Im registered transceivers, the Count loop counter, the flag multiplexers, and the main control logic. Figure 4-23 depicts in detail the components of the microprogrammed controller and the associated microprogram control fields.

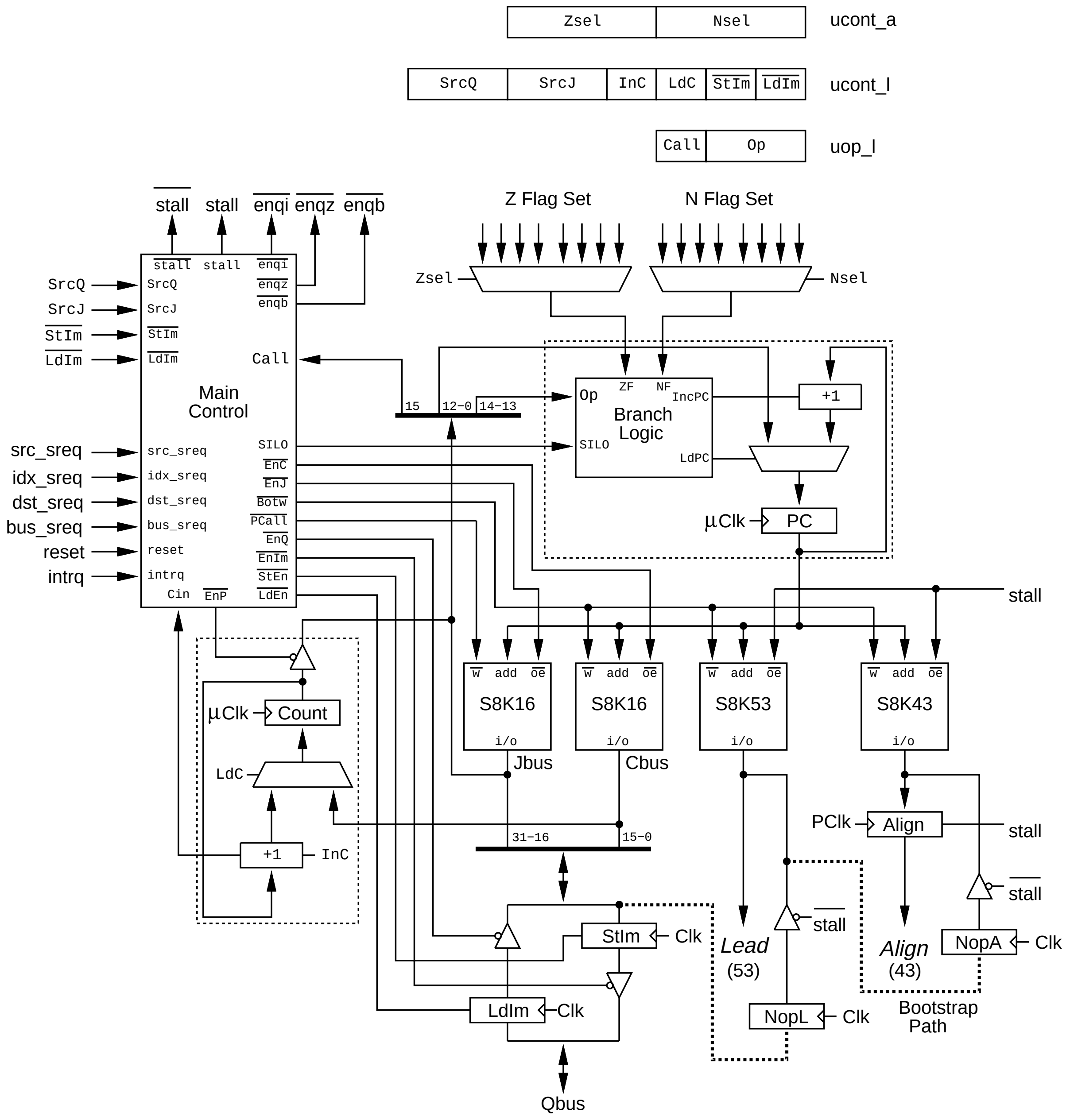


FIGURE 4-23: Microprogrammed controller and its associated microprogram control fields

The microprogram counter PC is a thirteen bit register that points to the current microinstruction being executed. The PC register can either be incremented or load an address from the thirteen least significant bits of the JBus under control of the branch logic. The branch logic takes as input the two bit SILO signals from the main control, the two bit Op opcode from uop_l (bits 13 and 14 in the JBus), and two generic flags ZF and NF. The SILO signals are interpreted as:

00 Stall. No change is allowed in the contents of the PC register.

01 Force an increment, regardless of Op or flag status.

10 Force a load from JBus, regardless of Op or flag status.

11 Operate. If the condition selected by the Op is true, load, else increment.

In the operate mode of the SILO, the Op microcontrol instruction will select a condition based on the contents of the ZF and NF flags. The Op specifies:

00 $\overline{\mathtt{NF}} \cdot \overline{\mathtt{ZF}}$

01 $\mathtt{NF} \cdot \overline{\mathtt{ZF}}$

10 $\overline{\mathtt{NF}} \cdot \mathtt{ZF}$

11 $\mathtt{NF} \oplus \mathtt{ZF}$

By carefully selecting the values of Zsel and Nsel in conjunction with Op it is possible to select various branch conditions that include but are not limited to the traditional $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$ floating point comparison operators, as well as detecting zero in the Count register, and testing the status of various bus interface FIFO signals. The PC register and its associated branch logic could be implemented with two *Cypress PAL22V10C* devices, but for speed it would probably require four devices using two identical sets, each driving half of the 96pF capacitive load in the microprogram memory address lines.

The microprogram memory consists of sixteen 8Kword by 8-bit fast common I/O SRAM devices, such as the *Cypress CY7B185*. These devices generate simultaneously the 128 control signals for the various processor subsystems. All SRAM devices share the same address lines, but their control signals are independently generated. The 128 memory data signals can be divided in three groups: the first group has sixteen bits connected to the JBus; the second group has sixteen bits connected to the CBus; and the third group contains the remaining 96 signals which can be further subdivided into 53 *Lead* and 43 *Align* control signals.

The NopL and NopA registers consist of 96 bits that will drive a programmable no op microinstruction to the various processor subsystems in the event of a processor stall. As shown in Figure 4-23, the stall signal will disable the memory output corresponding to *Align* and *Lead*, while enabling the NopL and NopA registers to drive these control lines. The input of these registers is tailored in such way that they form along with LdIm a four stage serial-to-parallel shift register that will be used during the microcode boot phase to hold the data to be written into the microprogram memory. During the microcode boot phase, data is loaded so that four 32-bit words coming serially from the QBus are shifted into a single 128-bit word that can be written simultaneously to the microcode memory by asserting both $\overline{\mathtt{PCall}}$ and $\overline{\mathtt{Botw}}$ write signals. While $\overline{\mathtt{Botw}}$ can only be asserted during the microcode boot phase, $\overline{\mathtt{PCall}}$ is activated any time a non-leaf procedure call is entered. Procedure calls will be further discussed later in this section. The last data written to the microprogram memory during the boot phase should be a no op microinstruction, as this is the value present in the NopL and NopA registers during the normal operation and will be microinstruction issued during a processor stall cycle. The NopL and NopA registers can be implemented with twelve *Texas Instruments 74ABT2952* devices.

The Align registers hold the set of forty-three *align* control signals, which must remain stable for the entire clock period. These registers are loaded every valid clock cycle, and they will hold the last valid value during processor stall cycles. These registers can be implemented with six *Texas Instruments 74ABT2952* devices.

The LdIm register can hold a branch address and instruction, along with a loop count. The register can be loaded from the QBus, and its data can be selected to drive simultaneously the JBus and CBus under microprogram control. The LdIm register can also be used during the microcode boot phase to shift in data from the external bus into the NopL and NopA registers. The StIm register is used to insert 32-bit immediate data into the QBus. The register holds data formed by juxtaposing sixteen bits from the JBus (upper word) and sixteen bits from the CBus (lower word). The 32-bit data can be driven by the microprogram memory or by other sources such as the LdIm register. These registers can be implemented with four *Texas Instruments 74ABT2952* devices.

Count is a sixteen bit register which can be used in "increment and branch on carry" operations for loop control, and for holding a branch address to be driven into the JBus for handling procedure calls. The Count register can be loaded from CBus or incremented under microprogram control, while the carry output can be used as a flag for conditional branches. The Count register and associated logic can be implemented with two *Cypress PAL22V10C* devices.

The two flag multiplexers are independently controlled by the `Zsel` and `Nsel` *align* microinstruction fields, which are part of the ucont_a microinstruction. The flag multiplexers can be implemented with two *Integrated Device Technology, Inc. 74FCT151CT* devices. The `Nsel` field selects as `NF` one of the following input signals:

000	False (always zero)
001	Floating point negative result flag
010	Count register carry out flag
011	Done output signal from the IIN FIFO ($\overline{\mathtt{do}}$)
100	Reserved
101	Reserved
110	Reserved
111	Reserved

The `Zsel` field selects as `ZF` one the following input signals:

000	False (always zero)
001	Floating point zero result flag
010	Count register carry out flag
011	Reserved
100	Empty output signal from the DIN FIFO ($\overline{\texttt{de}}$)
101	Full output signal from the DOUT FIFO ($\overline{\texttt{df}}$)
110	Empty output signal from the IIN FIFO ($\overline{\texttt{ie}}$)
111	Full output signal from the IOUT FIFO ($\overline{\texttt{if}}$)

The main control is a FSM that takes as input reset and intrq from the network interface, the four stall request signals generated by the various memory subsystems and bus interface, part of the ucont_I microinstruction, and the Call microinstruction from uop_I (JBus bit 15). This FSM uses these inputs and the internal state to generate the stall signal and its complement, various enable signals for QBus and JBus drivers, the SILO signal to control the microinstruction branch logic, the $\overline{\texttt{StEn}}$ and $\overline{\texttt{LdEn}}$ signals to enable respectively the load of the StIm and LdIm registers, and the microprogram memory write enable signals $\overline{\texttt{Botw}}$ and $\overline{\texttt{PCall}}$. The main control can be implemented with three *Cypress PAL22V10C* devices. The ucont_I specifies the following instructions:

$\overline{\texttt{LdIm}}$ Active low, it will enable data from the QBus to be loaded in the LdIm register in the rising edge of the Clk signal at the end of the following valid clock cycle by asserting the $\overline{\texttt{LdEn}}$ signal. The $\overline{\texttt{LdEn}}$ will also be asserted automatically by the main control FSM during the microprogram boot phase.

$\overline{\texttt{StIm}}$ Active low, it will enable data formed from the juxtaposition of JBus and CBus to be loaded in the StIm register in the rising edge of the Clk signal at the end of the following valid clock cycle by asserting the $\overline{\texttt{StEn}}$ signal. The main control will also use the SILO signal to force the microprogram counter to increment, ignoring the contents of JBus so that generic 32-bit data could be inserted in the StIm register from the microprogram memory.

LdC Active high, will load the Count register with the contents of the CBus in the rising edge of μClk at the end of the current valid clock cycle.

InC Active high, will increment the Count register in the rising edge of μClk at the end of the current valid clock cycle.

SrcJ Selects one of the following sources for JBus and CBus data during the next clock cycle:

00	LdIm register drives JBus and CBus ($\overline{\texttt{EnQ}}$ active)
01	Microprogram memory drives JBus and CBus ($\overline{\texttt{EnJ}}$ and $\overline{\texttt{EnC}}$ are active)
1x	Count register drives JBus, memory drives CBus ($\overline{\texttt{EnP}}$ and $\overline{\texttt{EnC}}$ are active)

SrcQ Selects one of the following sources for the QBus data during the next clock cycle:

00	IIN FIFO in the network interface drives QBus ($\overline{\text{enqb}}$ active)
01	StIm register drives QBus ($\overline{\text{EnIm}}$ active)
10	StIx register (index memory subsystem) drives QBus ($\overline{\text{enqi}}$ active)
11	ZBus FIFO (source memory subsystem) drives QBus ($\overline{\text{enqz}}$ active)

Figure 4-24 illustrates a sequence of microprogrammed control unit cycles that highlights the most relevant aspects of its functionality, and in special, the handling of procedure calls. Figure 4-24 provides a fuzzy idea of timing information as the windows in which data is stable are exaggerated, focusing on the sequencing aspects of the main control operation.

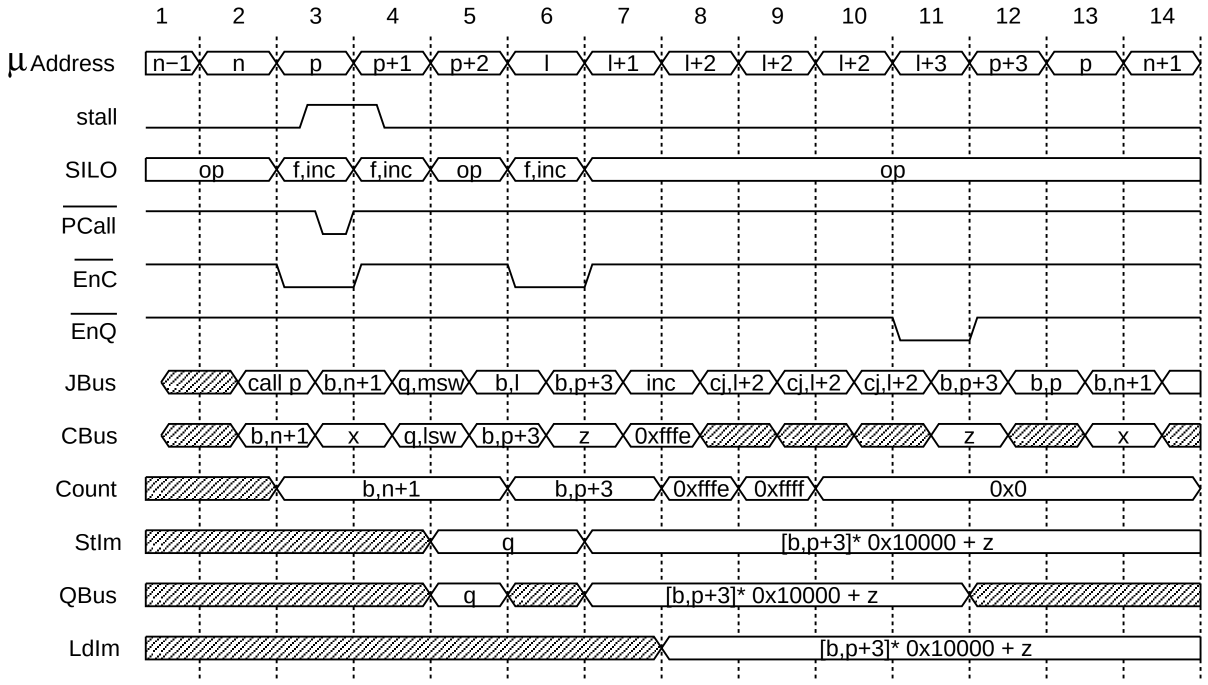

1 2 3 4 5 6 7 8 9 10 11 12 13 14
μAddress
n−1 n p p+1 p+2 l l+1 l+2 l+2 l+2 l+3 p+3 p n+1
stall
SILO
op f,inc f,inc op f,inc op
PCall
EnC
EnQ
JBus
call p b,n+1 q,msw b,l b,p+3 inc cj,l+2 cj,l+2 cj,l+2 b,p+3 b,p b,n+1
CBus
b,n+1 x q,lsw b,p+3 z 0xfffe z x
Count
b,n+1 b,p+3 0xfffe 0xffff 0x0
StIm
q [b,p+3]* 0x10000 + z
QBus
q [b,p+3]* 0x10000 + z
LdIm
[b,p+3]* 0x10000 + z

FIGURE 4-24: A relevant sequence of operations in the microprogrammed control unit

During the second cycle, the current uop_I instruction being executed at address n is a procedure call to the microcode address p. At the same time, sixteen bit immediate data that if present in the JBus would execute the instruction branch to address n+1 is stored in the Count register. During the third cycle, the main control actions in response to a procedure call request includes automatically generating stall signal for the next cycle, writing the contents of JBus into the microcode memory at address p, and forcing the microprogram counter to increment using the SILO signals. The system software is responsible for properly choosing the source for JBus during the third cycle, in this case with the contents of the Count register (or the microinstruction "branch to n+1"). During the fourth clock cycle, the SILO will again force the microprogram counter to increment, this time due to the $\overline{\text{StIm}}$ request during the execution of the previous instruction. The contents of JBus and CBus will be written into the StIm register (this data is labeled q), while the branch logic and the main control will ignore whichever values are present in the JBus.

During the fifth cycle, a *leaf* procedure call to address l is executed. The difference to normal procedure call is that the return address can be left in the Count register without writing it to the microcode memory, as a leaf procedure will never call other procedures. In this particular case, however, the Count register will be needed for a controlled loop. During the sixth clock cycle the $\overline{\texttt{StIm}}$ instruction is executed again, writing the pair { branch to address p+3, z } into the StIm register, which drives the QBus during the seventh cycle, and is read back into the LdIm register. During the seventh cycle the value -2 (0xfffe) is loaded into the Count register. The microinstruction at address l+2 is a conditional jump (cj, l+2) if no carry from the count register to the same address l+2, and it includes also an increment to the Count register. The loop instruction is repeated three times (as the Count register was initialized with -2). In the eleventh cycle, the contents of the LdIm register will drive JBus, executing a return instruction to address p+3, thus finalizing the execution of the leaf procedure.

In the twelfth cycle, a branch to first address p of the procedure is executed, which is the standard way in this system for returning from a non-leaf procedure. In the thirteenth cycle, the actual microcontrol instruction resident at p is executed, and the control returns to its caller by executing the "branch to address n+1" instruction that has been written at address p. Using this scheme for procedure calls, the number of nesting levels is limited only by the microprogram memory size, and no extra hardware is required. Stacks for procedure calls, which are difficult to implement with off-the-shelf parts are not necessary. A similar scheme was used in other machines, like the PDP-8. Perhaps a shortcoming of this method of procedure nesting is that recursive or reentrant code is not supported. This limitation is not very serious as there are only a small number of numerically intensive codes that use recursive procedure calls.

Figure 4-25 depicts in detail the most critical signal timing constraints that must be satisfied for the proper operation of the microprogrammed controller. Data is read from the microcode memory to JBus during at addresses u1 and u2, and is written to the microcode memory at address u3.

Table 4-4 summarizes the microprogrammed control system switching characteristics and timing requirements depicted in Figure 4-25 for the devices previously mentioned. Using the notation of Figure 4-25 and Table 4-4, and denoting the clock period T_{Clk}, the following constraints are the most difficult to be attained for the proper operation of the controller:

$$t_{\mu OD} + t_{AA1} + t_{S\mu} \leq T_{Clk} \tag{4.10}$$

$$t_{COD} + t_{AA1} + t_{SL} \leq T_{Clk} \tag{4.11}$$

$$t_{COD} + t_{AA2} + t_{SA} \leq T_{Clk} - t_{AOD} \tag{4.12}$$

Constraint 4.10 refers to the loop path from the PC register through the microprogram

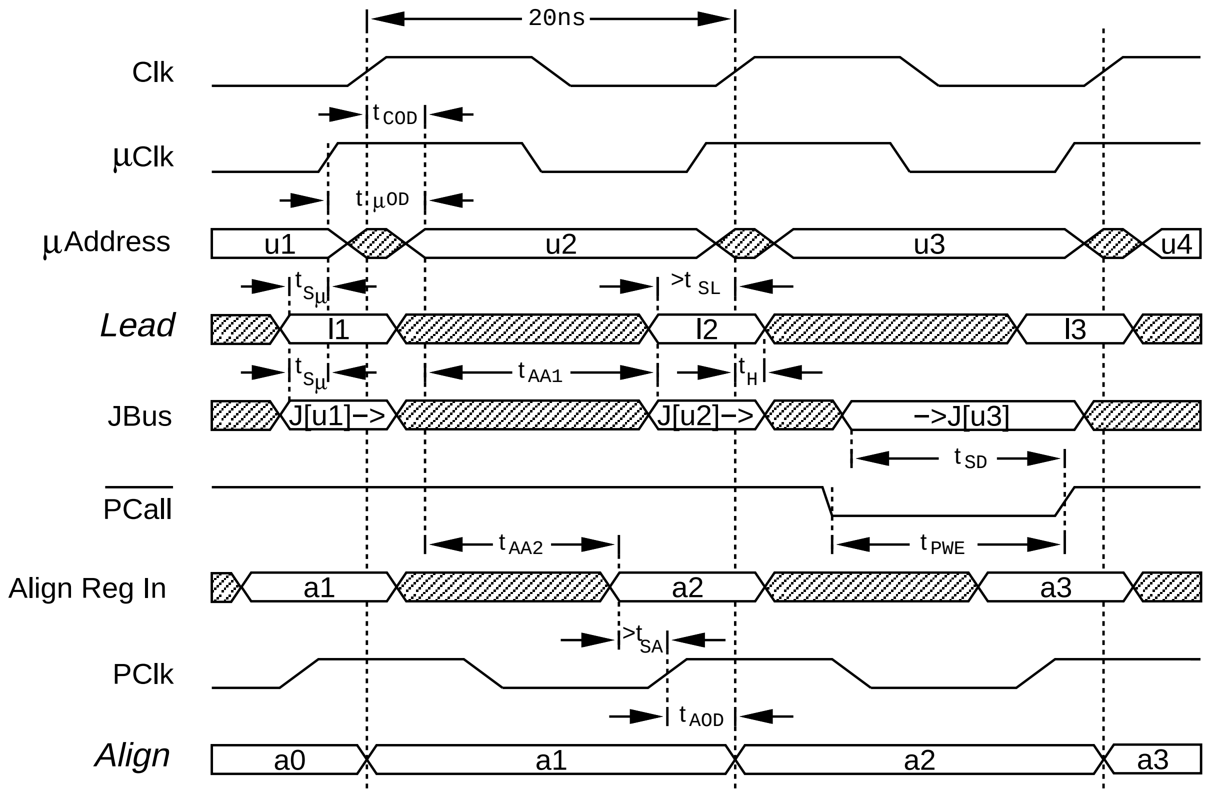


FIGURE 4-25: Microprogrammed controller system timing

Symbol	Description	Min(ns)	Max(ns)
t_{COD}	Clk to microaddress valid delay		4
$t_{\mu OD}$	μClk to microaddress valid delay		6
t_{AOD}	PClk to *Align* register output delay		5
t_{AA1}	Microprogram memory access time (slow)		10
t_{AA2}	Microprogram memory access time (fast)		9
t_H	Generic *Lead* signal hold time	2	
t_{SL}	Generic *Lead* signal setup time	4	
$t_{S\mu}$	Microprogram counter PC setup time	3	
t_{SA}	Align register setup time	2	
t_{SD}	Data setup time to write end	5	
t_{PWE}	Write pulse width	8	

Table 4-4: Microprogrammed controller switching characteristics and timing requirements

memory and back to guarantee a single cycle latency in all branch and loop instructions. Constraint 4.11 refers to the amount of skew between Clk and μClk in order to meet the minimum setup time requirements of all *lead* microcontrol signals in the other subsystems. Finally, the most difficult constraint to be satisfied is 4.12, which refers to the amount of delay from the rising edge of Clk through the microprogram address register PC, and then through the memory to stable data in the input of the Align registers, or the rising edge of PClk. For the proper operation of the system it is necessary first to adjust the skew $\Delta t(Clk, PClk)$ between PClk and Clk so that it matches the Align register output delay $\Delta t(Clk, PClk) = t_{AOD}$. The adjustment requires the usage of the techniques described in Section 4.1.1. The second step is the adjustment of skew $\Delta t(Clk, \mu Clk)$ between μClk and Clk so that both constraints 4.12 and 4.11 are satisfied.

Using extremely fast static memories [Chappell91] and VLSI techniques, it is possible to obtain a tenfold speedup in the microprogrammed control unit at the expense of increased operation latency.

The IC count for the microprogrammed control system is thirty-one 24-pin DIP packages for the *PALs* and registers, sixteen 28-pin DIP packages accounting for the fast SRAMs, and two 16-pin DIP packages accounting for the multiplexors. The estimated power dissipation is 27 Watts.

4.3 Architecture Emulation

The architecture emulation formally describes the system hardware. The objectives of such representation are documentation, global system verification, and design tradeoff analysis. The machine architecture emulation was performed at two levels, first at behavioral level, and then RTL (register transfer level). After a brief introduction of the behavior level emulation of the proposed multiprocessor architecture, we will discuss the RTL emulation.

4.3.1 Behavioral Level Emulation

Figure 4-26 illustrates the organization of the behavior level emulation of the multiprocessor. Each processor was modeled as having its own local index, source and destination memory, an input queue, and an output queue. The communications between processors are modeled by a queue representing the bus, and a list of message control vectors, which are discussed in Section 4.1.2. All the code executed by the target multiprocessor is written in C, compiled, and executed by the host machine performing the emulation.

A comprehensive simulation of the execution of `SIMLAB` (a circuit simulator similar to SPICE, but with much simpler device models) in a multiprocessor environment was executed at behavioral level. The objective of the simulation at this level was to prove the correctness of the proposed computation model both for the sparse matrix solution and the device evaluation phase. A modified version of `SIMLAB` would first parse the description of the circuit to be simulated, external input voltages and initial conditions, and reorder the associated sparse matrix in order to minimize fillin. A special precompiler using the techniques discussed in Chapters 2 and 3 partitions the sparse matrix and the device description, distribute their data into the memory image of the different "processors" in Figure 4-26, schedule the different tasks required for both

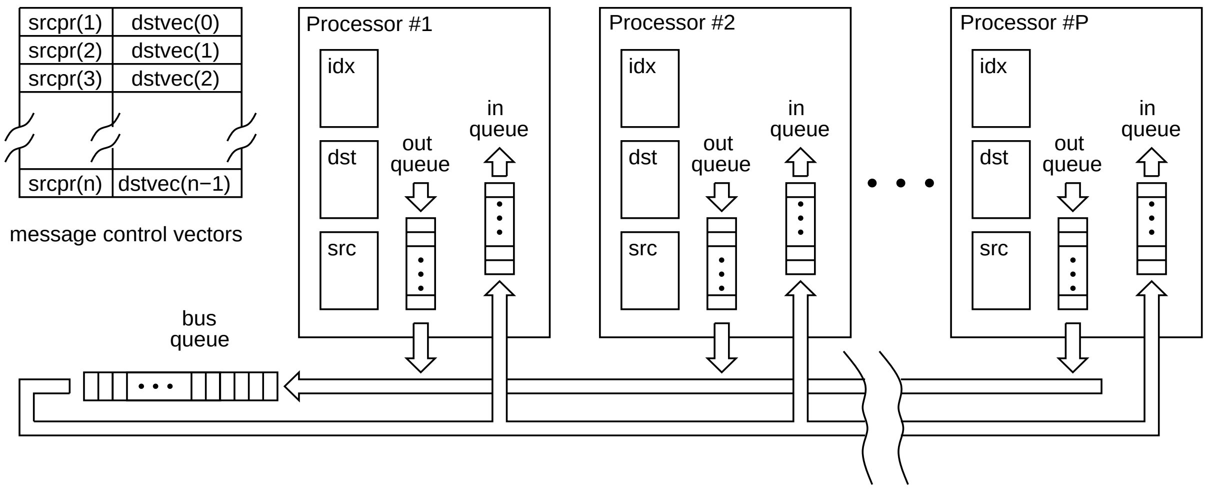


FIGURE 4-26: Behavioral level simulation of the proposed architecture

the device evaluation and sparse matrix solution in the form of a long list of task descriptors, load the task lists in the source and index memory of each "processor", and generate the list of message control vectors.

In the inner step of the Newton-Raphson iteration, described in detail in Chapter 3, SIMLAB would pass the control to the behavioral simulator. The simulation consists of visiting each "processor" and execute all tasks from the descriptor list until the list is exhausted, or there is an attempt to read from an empty input queue. After all processors are visited, the "bus processing" involves removing one message from the outgoing queue of the processor specified by the srcpr field of the next message control vector, and placing a copy of the message in the input queue of all the processors corresponding to set bits in the dstvec field of the message control vector. The loop is repeated until all the processors have exhausted all the tasks in the descriptor lists. After all processors have executed all tasks, the control returns to SIMLAB.

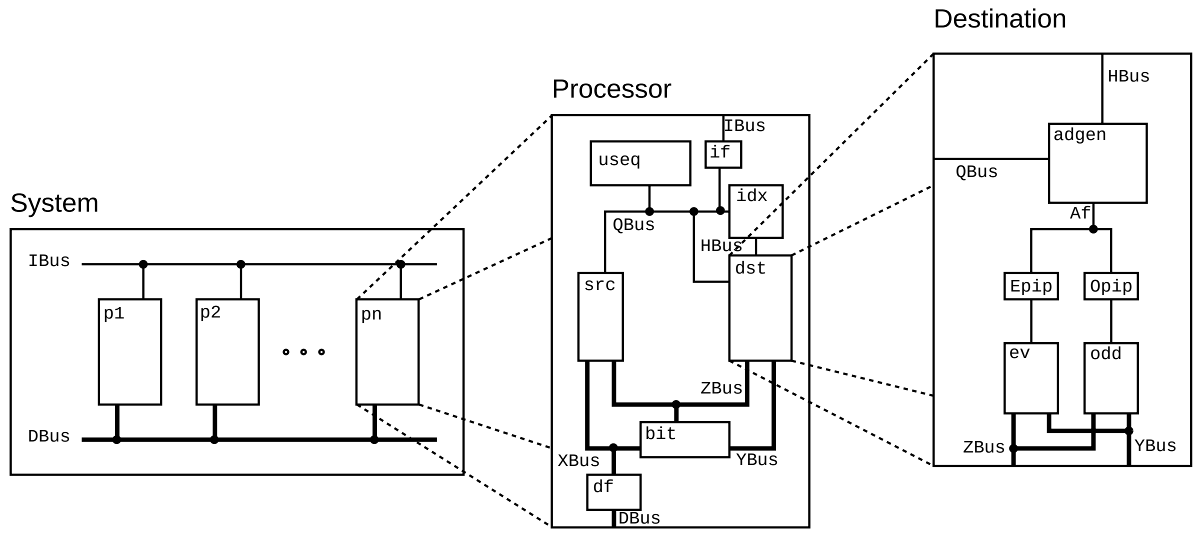
The perfect match between the output results of the modified for behavioral level simulation version of SIMLAB and the original version of SIMLAB proves the correctness of the partition, scheduling, and message passing schemes, as well as the correct generation of the local processor memory addresses. The results from the behavioral level emulation open the path for a more detailed emulation of the architecture, using the RTL model.

4.3.2 RTL Emulation

The RTL model describes the flow of data between registers and how data is modified in pure combinational logic. We are initially interested in obtaining accurate information on the number of clock cycles required for the completion of some computational task. Accurate timing information is not essential for this model, provided that we limit the maximum amount of delay in a combinational logic block to some value that is consistent with our target physical clock period.

The basic idea behind the RTL simulation model is the representation of relevant internal registers in the form of a consistent data structure, and the activation of a procedure that represents the rising edge of Clk. The procedure execution will update all the registers in a particular block by evaluating a function of the previous state and external inputs. This model is a valid representation of the actual hardware provided that the clock period is larger than $t_{CO} + t_{PD} + t_{SU}$, where t_{CO} represents the register's clock to output delay, t_{PD} is the maximum propagation delay through the combinational logic, and t_{SU} is the register setup time.

Modularity and hierarchical representation of the system were the key issues in the software design. The usage of these concepts greatly simplify the programming, help the documentation, and eases the debugging. Each module consists of a *data structure*, and a set of *procedures*, which represent its structure and functionality. Figure 4-27 depicts the hierarchical organization of the hardware and a simplified version of the C language data structures that represent the internal organization of each module.



```
typedef struct {

  int         IBus;
  double      DBus;
  int         n;
  Processor   *proc_array[];

  } *System;
```

```
typedef struct {

  int            QBus;
  int            HBus;
  double         XBus;
  double         YBus;
  double         ZBus;
  Source         *src;
  Destination    *dst;
  Index          *idx;
  DFifo          *df;
  IFifo          *if;
  Fpu            *bit;
  Microseq       *useq;

  } *Processor;
```

```
typedef struct {

  int        Af;
  Address    *adgen;
  Pipeline   *Epip;
  Pipeline   *Opip;
  Memory     *ev;
  Memory     *od;

  } *Destination;
```

FIGURE 4-27: Modular hardware representation of the numerical engine and its corresponding software data structures

In this representation, not only the registers, but also tri-state busses and some "debug" internal wires are all part of the data structure definition. Tri-state bus data must be kept across procedure calls in order to represent its physical behavior — when a bus is not driven, it keeps the last data for a certain period of time. The basic set of procedure calls that are connected with a certain module are: `make_module()`, which allocates the data structures; `clk_module()`, which actually performs the RTL simulation; and a set of `particular_output_module()` procedures, which are used for setting the state of the external interfaces before any calls to `clk_module()` are made.

Each module representing a block previously discussed in this chapter is contained in a `module.c` and a `module.h` file. All modules are linked together and with the modified version of the `SIMLAB` program which contains the precompiler for partitioning, scheduling, address and microcode call generation. The precompiler also binds the internal code generation database with microcode external symbols to determine microcode branch destinations. Also implemented is a mechanism that mimics microcode boot, and a microcode debugger. This organization, along with the microcode emulation of a memory-mapped device in the actual hardware would allow, in the future, the usage of this tool with minimal modifications for debugging the actual hardware.

In order to simplify the development of the microcode, and the binding of microcode symbols with the scheduler and code generation mechanism inside the modified version of `SIMLAB`, a rather sophisticated microassembler with relocation and external symbol handling capabilities was written. The relocatable object code generated by the microassembler is linked and loaded into the microcode memory of all processors modeled at RTL level prior to the execution of the RTL emulation. An example of the microassembler code for the evaluation of the exponential function e^x is provided in Appendix A.

Three thousand lines of microassembler code corresponding to some three hundred microinstructions were written and fully debugged for the implementation of the solver for sparse sets of linear equations. The results were checked by introducing random solution vectors, multiplying them by the original matrix to obtain the right hand side vectors, compute the solution of the sparse set of linear equations, and compare the solution with the original randomly generated vectors. The relative norm of the errors obtained were in the order of 1×10^{-13}. The results of the architecture emulation of the sparse matrix solver in a single processor using the RTL model, and a comparison with general purpose platforms commercially available today are described in detail in the next section.

4.4 Architecture Emulation Results

Table 4-5 compares for the test matrices described in Chapter 2 the sparse matrix factorization times measured on several general purpose machines and the RTL simulation results for a single processing element of the Numerical Engine described in this chapter. All times

matrix	SPARC2 40Mhz	MIPS3000 40Mhz	RS6000/540 50Mhz	AXP ALPHA 150Mhz	Numerical Engine (RTL) 50Mhz
dram	5300	3511	780	666	26
feb	867	836	340	300	22
iir12	3433	3676	1430	1350	53
iir123	5033	5410	2030	1917	78
mesh	567	559	180	183	10
omega	217	215	90	67	7
mfr	817	809	290	267	19
Average	742	728	274	232	17

Table 4-5: Sparse matrix factorization times

are in miliseconds. The code executed in the other architectures is the sparse matrix package Sparse 1.3b [Kundert88]. The Numerical Engine results are obtained by multiplying the total number of cycles necessary to decompose the sparse matrix by the 20ns clock period, using the scheduling, memory allocation and data reordering algorithms discussed in Chapter 2. Table 4-6 presents the same results for processor speed in terms of million of floating point operations per second (MFlops).

The measurements for the SPARC2 and MIPS3000 are very similar, as their architectures have comparable performance and they operate at the same clock speed. These processors have a rather slow floating point unit, and the simple scalar integer architecture poses a large overhead for address calculation and floating point data access.

The comparison between the performance of the RS6000 and the AXP ALPHA is rather puzzling. Both processors have a very high performance floating point unit that can execute one floating point multiply or add per clock cycle, and the advanced superscalar (or superpipelined for the ALPHA) architecture can reduce substantially the data access overhead by executing several instructions concurrently. However, instead of an expected 3$\times$ speed up for the AXP ALPHA due to the clock speed ratio, empirical data shows speedups in the order of 15% only. This effect seems to be related with the main memory bandwidth. Since both systems have a similar main memory structure, and use DRAMs devices of the same speed, the results from Table 4-5 and 4-6 help support the hypothesis that the main memory bandwidth is the performance bottleneck for sparse matrix decomposition in general purpose machines. The largest improvement for the AXP ALPHA, which occurred for the *omega* test case, might also be explained by a better utilization of the cache mechanism, as *omega* is one of the smallest test matrices.

The Numerical Engine architecture overcomes the memory bandwidth bottleneck by providing simultaneous access to index, source, and destination data through different paths and accelerating these accesses by interleaving. The RTL simulated results show that at least one order of magnitude speed improvements over the other architectures can be achieved for the test matrices discussed. The memory address computation overhead has also been reduced by the usage of concurrent address computation units. The smallest speed improvements obtained, for *omega* and *feb* matrices, are probably linked with the task initialization overhead, given the relatively small average size of the row-wise tasks for these two matrices and the relatively large pipeline latency.

The predicted results are very encouraging. Even if the target clock period of 20ns cannot be achieved, the combined usage of the hardware and software techniques discussed in this thesis are still expected to provide a large amount of speedup compared with other architectures.

matrix	SPARC2	MIPS3000	RS6000/540	AXP ALPHA	Numerical Engine
Peak	4MFlops	4MFlops	50MFlops	150MFlops	(RTL)100MFlops
dram	0.36	0.54	2.43	2.85	73.0
feb	0.98	1.01	2.49	2.82	38.5
iir12	1.07	1.00	2.58	2.73	69.6
iir123	1.04	0.97	2.58	2.73	67.0
mesh	1.21	1.23	3.81	3.74	68.5
omega	0.98	0.99	2.36	3.18	30.4
mfr	1.12	1.13	3.14	3.41	48.0
Average	0.97	0.98	2.77	3.06	56.4

Table 4-6: Sparse matrix factorization performance (MFlops)

5

Conclusion and Future Work

5.1 Major Contributions

This thesis presents hardware and software techniques for the fast numerical solution of differential equations, focusing on the execution of a modified version of a circuit simulation program, SIMLAB. The components that comprise most of SIMLAB execution time are the assembly of the circuit equations and its associated sparse matrix solution.

In SIMLAB, the assembly of circuit equations involves the evaluation of the circuit devices and the stamping of their contributions in the Jacobian matrix and in the right-hand-side current and charge vectors. While the device evaluation can be done independently in different processors, the stamping of their contributions requires either the replication of the device evaluation effort or the transmission of the contributions across the network. For linear devices and voltage sources, we have proposed a scheme that divides these devices into *local* and *shared*. Since the evaluation cost for these devices is usually very small, *shared* devices are replicated in different processors. In the case of non-linear devices, if task replication is used, the empirical results in Table 3-1 show that the multiprocessor utilization tends to 50% as the number of processors grow. In contrast, if the contributions for stamping are transmitted over the network, the multiprocessor utilization will be around 90%, provided the network is not saturated. Given simple cost functions, the behavioral level emulation of the Numerical Engine shows that the bus saturation happens when the number of processors $P \approx 20$.

This thesis introduces the O^2SA technique, which combines scheduling and storage allocation algorithms for the fast parallel sparse matrix decomposition. The essence of this technique is to keep in the cache an active data subset, necessary to attain a high degree of parallelism. The superiority of the O^2SA scheme in terms of multiprocessor utilization over the Scatter-Gather technique becomes evident by comparing Figures 2-13 and 2-14. In fact, the degree of multiprocessor utilization using the O^2SA technique is comparable with the utilization that can be attained using the OSA technique, even with a modest cache size. The major advantage of the O^2SA technique over OSA is the great speed improvement achievable in each processing element. The results of register transfer level (RTL) emulations of a single processor presented in Table 2-9 indicate that the O^2SA-based hardware can deliver on average 2.6× the speed of an OSA-based hardware with comparable technological constraints by properly exploiting the *locality of reference*.

It is possible to exploit the properties of sparse matrices and scattered arrays by reordering the data to efficiently use memory interleaving as a general technique for increasing processor-memory communication bandwidth. Besides the obvious improvement in the sequential access to the source and index elements in a row update operation, it is possible to achieve up to 90% interleaving efficiency in the access of a 2-way interleaved memory holding destination data, as shown in Table 2-2. The RTL emulation results presented in Table 2-3 indicate that the number of static column misses are reduced by reordering on average 5×. The total number of stalled cycles was reduced to a third of the original figure, yielding global improvements in the order of 25%.

The Numerical Engine has specialized datapath and control in each processing element that permits the execution of several tasks per clock cycle. During the sparse row update execution, the most frequent operation in sparse matrix decomposition, ten operations are started in each processing element per clock cycle — three integer operations, two floating point operations, four memory accesses, and a conditional jump. These operations are described in detail in Section 2.5.2. Experimental data presented in Table 4-5 and Table 4-6 indicates that the bandwidth between the main memory and the floating point unit is the performance bottleneck for sparse matrix decomposition in general purpose machines. For instance, the AXP Alpha platform, a state-of-the-art superscalar computer with a peak performance of 150 MFlops, achieves on average only 3 MFlops during the sparse matrix decomposition, corresponding to a floating point unit utilization of 2%. The Numerical Engine architecture overcomes the memory bandwidth bottleneck by providing simultaneous access to index, source, and destination data through different paths and accelerating these accesses by interleaving. The RTL simulations for a single Numerical Engine processor show at least one order of magnitude speed improvement over the other architectures during the execution of the sparse matrix decomposition. Correspondingly, the utilization of the floating point unit in the Numerical Engine is very high, up to 73% in the case of the *dram* matrix.

5.2 Future Work

Perhaps the most serious shortcoming of precompilation techniques for sparse matrix decomposition is related with numerical pivoting. The robustness of a circuit simulator is strongly connected with its ability to perform numerical pivoting, which consists of reordering rows and/or columns of the matrix, when a pivot, also called divider in a row normalize operation, becomes too close to zero in the course of the simulation. With the current precompilation scheme, if such event happens, it would be necessary to stop the decomposition, flush all matrix data from all processors and restart again all the precompilation steps to deal with a new symbolic matrix. An interesting research topic is an extension of these precompilation techniques to accommodate alternate paths that would be executed in these situations, thus avoiding a new precompilation step.

Another important research path is the extension of the precompilation techniques discussed in this thesis for general purpose parallel processors, involving the implementation of O^2SA-based schemes on sequential machines and the implementation of the overall multiprocessor scheme in a general-purpose parallel machine. As discussed in Section 2.5, the O^2SA storage and scheduling scheme can be applied with some modifications to a system with a block-oriented main memory system (DRAM), and a small cache (SRAM), which are found on most computers today. The implementation of a new sparse matrix package based on the O^2SA strategy might improve substantially the floating point utilization of modern general purpose computers. Also, as discussed in Section 3.3, most general purpose parallel computers available today have a long message latency, and a large overhead in order to set up a message. Even if the message latency and setup times could be shortened, it is unlikely that the ratio between the latency for message passing and the floating point unit bandwidth will change substantially for general purpose multicomputers. Given these constraints, the sparse matrix factorization code will still be very difficult to parallelize on a general purpose machine, as it is very difficult to find a way of partitioning the data amongst the processors to properly balance the load and at the same time to minimize the impact of the network latency on the critical path of the solver.

A

Fast Evaluation of Transcendental Functions

A.1 Background

We will describe in this appendix the implementation of efficient routines for the evaluation of transcendental functions. The algorithm chosen for these functions is the Chebyshev series expansion method [Clenshaw63].

There are several reasons to choose Chebyshev's method rather than other more familiar methods, like Taylor series and Newton-Raphson successive approximation. The primary advantage of Chebyshev's method is that it exhibits uniform convergence in a specified number of terms, so that the range of input value will have little influence on the accuracy of the result. Another advantage is that it is an economical method, in the sense that the number of terms required to achieve a given precision is relatively small, which in turn provides for fast execution. Also, the bounds of the error are easily accessed. Lastly, it is widely applicable, as any continuous function of bounded variation has a convergent Chebyshev series expansion.

The infinite Chebyshev series for the function $f(x)$ in the range $-1 \leq x \leq 1$ takes the form

$$f(x) = \frac{1}{2}c_0 + c_1T_1(x) + c_2T_2(x) + \ldots \tag{A.1}$$

where $T_r(x)$ is the Chebyshev polynomial of degree r, defined by

$$T_r(x) = \cos\,(r \arccos x) \tag{A.2}$$

and c_r is the Chebyshev coefficient. By truncating the series (A.1) after the term $c_nT_n(x)$ we obtain an approximation to $f(x)$ of the n-th degree. Since $|T_r(x)| \leq 1$ for all r, the approximation error cannot exceed $\sum_{r=n}^{\infty} |c_r|$, for which a bound can be obtained by inspection.

The form shown in (A.2) is quite cumbersome to use directly. An easier way of computing the Chebyshev polynomials of a degree r is by using the following properties:

$$\begin{aligned} T_{r+1}(x) - 2T_r(x) + T_{r-1}(x) &= 0 \\ T_0(x) &= 1 \\ T_1(x) &= x \end{aligned} \tag{A.3}$$

Sometimes it is better to evaluate the Chebyshev series in terms of polynomials $T_r^*(x)$. Since $T_r^*(x) = T_r(2x-1)$ we have only to replace x by $2x-1$ in Equation (A.1). In this case, the range of x is restricted to $0 \leq x \leq 1$.

Other cases of Chebyshev series of common occurrence are the even and odd series, in which the alternate coefficients vanish. Although we could use (A.1) directly, it is easier, using the properties of Chebyshev polynomials, to simultaneously replace x by $2x^2-1$ and c_r by c_{2r}, and writing:

$$T_{2r}(x) = T_r(2x^2-1) \tag{A.4}$$

Odd functions can be dealt by first removing the factor x. This factor is replaced after the evaluation of the resulting even function. This simple modification has the incidental advantage of preserving the accuracy for x very close to zero.

With these properties in mind, we can devise a simple method for computing the value of $f(x)$ for any given x. The routine first transforms the argument in a manner appropriate to the function in order to obtain a new variable which lies in the standard range $[-1,1]$ (or sometimes $[0,1]$). The Chebyshev series, truncated to the degree n depending on the accuracy desired, is then evaluated for the given value of the new variable. Tables containing the Chebyshev coefficients for a wide range of trigonometric, exponential, hyperbolic and many other functions found in standard mathematical libraries are available [Clenshaw62].

The computation of the finite series may be evaluated in two ways. The first, is by recurrence directly from the Chebyshev coefficients c_i. We form successively $b_n, b_{n-1}, ..., b_0$, from

$$b_r = 2xb_{r+1} - b_{r+2} + a_r, \qquad b_{n+1} = b_{n+2} = 0 \tag{A.5}$$

and then $f(x) = \frac{1}{2}(b_0 - b_2)$, a result that can be proved with the aid of (A.3). The second, is by reordering the series in the form

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_nx^n \tag{A.6}$$

Even though the literature presents many arguments in favor of the recurrence (A.5) directly from the Chebyshev coefficients [Clenshaw62], the form (A.6) is computationally cheaper if the series is always truncated to a fixed n. The coefficients a_r can be precomputed only once, and stored for posterior usage. The direct application of Horner's Rule to (A.6) yields

$$f(x) = (((((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + ... + a_2)x + a_1)x + a_0 \tag{A.7}$$

The form (A.7) is inefficient for a floating point unit with concurrent adder (ALU) and multiplier(MPY). The inefficiency is caused by the interdependencies of add and multiply operations: each add requires the result from the previous multiplication and vice-versa. Assuming, in the best case, that the latency is one cycle for both multiply and add, the FPU utilization is only 50%. By rearranging (A.7) in even and odd terms we can write, assuming n even,

$$f(x) = (((a_n x^2 + a_{n-2})x^2 + a_{n-4})x^2 + ... + a_2)x^2 + a_0 + \tag{A.8}$$

$$(((a_{n-1} x^2 + a_{n-3})x^2 + a_{n-5})x^2 + ... + a_1)x \tag{A.9}$$

The same type of reorganization can be done for n odd. In any case, we keep x^2 multiplying successively the previous results of the ALU, while the ALU adds the previous results of the MPY to a new input coefficient. The net result is an interleaved multiply-add operation: in a given cycle, the ALU handles an even term and the MPY an odd term, and in the next cycle they switch roles. This mechanism takes full advantage of floating point hardware with separate functional units. We shall describe in detail this operation through an example for the Numerical Engine in Section A.2.

Even though tables containing the values of the Chebyshev coefficients c_r are available, the task for obtaining a_i from c_i is tedious. This task can be automated using Algorithm A.1. This algorithm has been implemented in bc language, which provides arbitrary precision arithmetic.

The algorithm builds iteratively the T_i polynomial coefficients $T_i[j]$ using (A.3), multiplies these coefficients by c_i and accumulate in $a[j]$, the corresponding coefficient of x^j in Equation (A.6).

A.2 Evaluation of Transcendental Functions in Multiple Functional Units

We shall demonstrate the usage of the Chebyshev series expansion in a real example, the evaluation of the exponential function e^x, which also illustrates the interaction of the algorithm with the floating point hardware. In order to efficiently compute e^x we intend to use the Chebyshev Series expansion, properties of IEEE floating point numbers, and reorganization of the polynomial evaluation, as described by Equations (A.8) - (A.9).

We start by transforming the problem of computing e^x into a problem of computing 2^y, using

$$e^x = 2^{x \log_2 e} = 2^y, \quad if \quad y = x \log_2 e \tag{A.10}$$

Algorithm A.1 (Obtains Polynomial Coefficients from Chebyshev).

```
                /* $c$ vector contains input Chebyshev recurrence coefficients */
$a$[0]= $c$[0]/2
$a$[1]= $c$[1]
$T_{r-1}$[0]= 1
$T_r$[0]= 0
$T_r$[1]= 1
$i$= 2
while $i \leq n$ {
   $j$= 0
   while $j < (i-1)$ {
      $T_{r+1}$[$j$]= -$T_{r-1}$[$j$]                          /* copy $T_{r-1}$ into $T_{r+1}$ */
      $j$= $j$ + 1
      }
   $T_{r+1}$[$j$]= 0
   $T_{r+1}$[$i$]= 0
   $j$= 0
   while $j < i$ {
      $T_{r+1}$[$j$+1]= $T_{r+1}$[$j$+1] + 2 * $T_r$[$j$]                 /* add $2T_r$ */
      $T_{r-1}$[$j$]= $T_r$[$j$]                              /* copy $T_r$ into $T_{r-1}$ */
      $j$= $j$+1
      }
   $j$= 0
   while $j \leq i$ {
      $a$[$j$]= $a$[$j$] + $c$[$i$] * $T_{r+1}$[$j$]                  /* accumulate $c_iT_i$ */
      $T_r$[$j$]= $T_{r+1}$[$j$]                              /* copy $T_{r+1}$ into $T_r$ */
      $j$= $j$+1
      }
   $i$= $i$ + 1
   }
                        /* $a$ vector contains output polynomial coefficients */
```

In order to reduce the range, we can write $y = \ N - t$, where $N = \lceil y \rceil$. In this case, $e^x = 2^N 2^{-t}$, and $t \leq 1$. t is the new argument of the Chebyshev series expansion, in terms of $T^*_r(x)$. 2^N can be easily computed, using the IEEE representation of floating point numbers, by shifting $1023 + N$ to the exponent field of a double precision floating point number. In fact, the *Bipolar Integrated Technology, Inc. BIT 2130* FPU hardware used in the Numerical Engine provides a scale operation, which computes directly 2^N, given N. We are left to compute 2^{-t} using the Chebyshev Series expansion

Cycle	μAddress	ALU	MPY	XBus	YBus
	previous	previous	previous	x	$\log_2 e$
1	exp	-	$y = x\log_2 e$	2.0	-
2	exp_1	$N = ceil(y)$	$2y$	1.0	-
3	exp_2	$2y+1$	$2N$	-	-
4	exp_3	$z = 2N - 2y - 1$	-	-	a_9
5	exp_4	$int(N)$	z^2	1.0	a_{11}
6	exp_5	2^N	$o = z^2 a_{11}$	-	a_{10}
7	exp_6	$o = o + a_9$	$e = z^2 a_{10}$	-	a_8
8	exp_loop	$e = e + a_8$	$o = z^2 o$	-	a_7
9	exp_loop	$o = o + a_7$	$e = z^2 e$	-	a_6
10	exp_loop	$e = e + a_6$	$o = z^2 o$	-	a_5
11	exp_loop	$o = o + a_5$	$e = z^2 e$	-	a_4
12	exp_loop	$e = e + a_4$	$o = z^2 o$	-	a_3
13	exp_loop	$o = o + a_3$	$e = z^2 e$	-	a_2
14	exp_loop	$e = e + a_2$	$o = z^2 o$	-	a_1
15	exp_loop	$o = o + a_1$	$e = z^2 e$	z	a_0
16	exp_8	$e = e + a_0$	$o = zo$	-	-
17	exp_9	$2^{-t} = e + o$	-	-	-
18	exp_10	-	$e^x = 2^N 2^{-t}$	-	-

Table A-1: Computation of e^x on a FPU with concurrent ALU and MPY

$$2^{-t} = \frac{1}{2}c_0 + \sum_{r=1}^{\infty} c_r T_r^*(t) \tag{A.11}$$

By using $T_r^*(x) = T_r(2x-1)$, we replace t by z such that $z = 2t - 1$, or $z = 2N - 2x\log_2 e - 1$. We have chosen to use eleven terms of the Chebyshev series $c_{10}...c_0$, in order to obtain a maximum error smaller than 2×10^{-16}, which is the IEEE 754 double precision floating point machine precision. The eleven Chebyshev coefficients c_i from a table [Clenshaw62] were transformed to a_i using Algorithm A.1. The numerical value of these coefficients is listed in the `data` section of the microcode shown in Figure A-1. We are left with the evaluation of

$$2^{-t} = (...(((a_{11}z + a_{10})z + a_9)z + ...) + a_1)z + a_0 \tag{A.12}$$

or, applying the result in (A.8 - A.9)

$$\begin{aligned} 2^{-t} &= (...((a_{11}z^2 + a_9)z^2 + ...)z^2 + a_1)z + \\ &\quad (...((a_{10}z^2 + a_8)z^2 + ...)z^2 + a_2)z^2 + a_0 \end{aligned} \tag{A.13}$$

```
/*****************************************************************
***                                                            ***
***   exp.u  contains the procedure for solving y= exp(x)      ***
***   Given x, the system computes exp(x) in 18 machine cycles ***
***   (360ns at 50Mhz).                                        ***
***                                                            ***
***   Author: Ricardo Telichevesky                             ***
***           RLE - LCS - Massachusetts Institute of Technology ***
***           Version 3.0 of Jan 04, 1993                      ***
***                                                            ***
***   input: everything should be set in such a way that the xbus ***
***          will contain x (the input) and the ybus will contain ***
***          log2e. Also, it is required the request for imm32 ***
***          data in the jbus/cbus (stim). Also, in the call inst ***
***          one should have xbus <- fregs because in _exp+1   ***
***          we use it. Of course, normal leaf procedure call  ***
***          organization applies here                         ***
***                                                            ***
***   output: mul contains y= exp(x)                           ***
***                                                            ***
*****************************************************************/

bind    _exp                    /* bind program entry point */

number_iterations= 8-1          /* internal loop counts 8 times */

text

_exp:

        imm32 sram_exp_constants
        mx <- xbus, my <- ybus
        mop, dmult              /* m1 <- xlog2e (=y) */
        xregsel _register_two
        stim                    /* request next cycle jbus/cbus imm32 */
        qbus <- im              /* next cycle qbus <- sram_exp_constants */
        jbus <- count           /* next cycle jbus comes from count register */
        xbus <- fregs           /* two cycles from now xbus <- fregs */

exp_1:
        cont                    /* ignored */
        aop, dffi
        mx <- xbus, my <- mul   /* m2 <- m1 * 2.0 (=2y) */
        ax <- mul               /* a2 <- double ceil(m1) (=N) */
        xregsel _register_one
        enal, passl             /* next cycle execute af= al(=sram_exp_const) */
        qbus <- im              /* next cycle qbus <- return address */
        ldim                    /* in the end of next cycle ldim <- ret addr */

exp_2:
        cont
        ax <- xbus, ay <- mul   /* a3 <- (1 + 2m1) (=2y+1)*/
        aop, dadd
        my <- alu,              /* m3 <- 2a2  (=2N) */
        areg <- alu             /* areg <- a2 (=N) */
        enaf, incf              /* next cycle execute af++ */
        dr                      /* next cycle start dest read cycle (c9) */

exp_3:
        cont
        ax <- mul, ay <- alu    /* a4 <- m3 - a3 (=2N-2y-1=z) */
        aop, dsub
        xbus <- fregs           /* two cycles from now xbus <- fregs */
        enaf, incf              /* next cycle execute af++ */
        zreg <- alu             /* zreg <- a4 (=z) */
        dr                      /* start read cycle (c11) */

exp_4:
        cont
        ax <- areg
        aop, dfcsi              /* a5 <- int ceil(a2) (=int(N)) */
        mx <- alu, my <- alu    /* m5 <- a4 * a4  (=z^2) */
        areg <- ybus            /* areg <- c9  */
        xregsel _register_one
        enaf, incf              /* next cycle execute af++ */
        dr                      /* start read cycle (c10) */
```

```
exp_5:
        cont
        ax <- xbus, ay <- alu     /* a6 <- 1.0 * 2**a5 (=2^N) */
        aop, dscale
        mx <- mul, my <- ybus     /* m6 <- c11 * m5 (=c11*z^2=odd) */
        ldz                       /* srcz register <- a4 (=z) */
        enaf, incf                /* next cycle execute af++ */
        dr                        /* start read cycle (c8) */

exp_6:
        cont
        csel -number_iterations   /* set up to count number_iterations */
        cld
        ax <- areg, ay <- mul     /* a7 <- m6 + c9  (=c9+odd=odd) */
        aop, dadd
        my <- ybus                /* m7 <- c10 * m5 (=c10*z^2=even) */
        areg <- alu               /* areg <- a6 (=2^N) */
        enaf, incf                /* next cycle execute af++ */
        dr                        /* start read cycle (c7) */
        zflag <- tc               /* don't forget that ucont_a applies to next */

exp_loop:
        bne exp_loop, zflag <- tc /* do this number_iterations */
        cinc
        ax <- mul, ay <- ybus     /* a14 <- m13 + c2  (=c2+even=even) */
        aop, dadd
        my <- alu                 /* m14 <- a13 * m5 (=odd*z^2=odd) */
        xbus <- srcz              /* two cycles from now xbus <- srcz */
        enaf, incf                /* next cycle execute af++ */
        dr                        /* start read cycle (c0) */


exp_8:
        cont                      /* will continue because force false cond */
        ax <- mul, ay <- ybus     /* a16 <- m15 + c0 (=c0+even=even) */
        my <- alu, mx <- xbus     /* m16 <- a15 * a4 (=odd*z=odd) */

exp_9:
        cont
        ax <- alu, ay <- mul      /* a17 <- a16 + m16 (=even+odd=2^-t)*/
        jbus <- im                /* next cycle return instruction is executed */

exp_10:
        cont                      /* bogus - not used it is actually a branch */
        mx <- alu, my <- areg     /* m18 <- a17 * a6 (=2^(-t)*2^N=y= exp(x))*/
        zreg <- mul               /* in the next cycle z reg has output */



/* these are the constants that are resident on SRAM */

ddata

sram_exp_constants:
        word 0xbde351b8 0x21ac16d5  /*   -1.4056579584000002e-10     (a9)    */
        word 0xbd45a7fc 0x05d3b501  /*   -1.5387648e-13              (a11)   */
        word 0x3d957bfd 0x2dbf487c  /*    4.8849715200000002e-12     (a10)   */
        word 0x3e2f5b0e 0x17440879  /*    3.6502820159999996e-09     (a8)    */
        word 0xbe769e51 0xee631e87  /*   -8.4260246421760012e-08     (a7)    */
        word 0x3ebc8d75 0x30548dd5  /*    1.70186575704992e-06       (a6)    */
        word 0xbefee4fd 0x234a4926  /*   -2.9463279320804e-05        (a5)    */
        word 0x3f3bdb69 0x6e8987ac  /*    0.000425065269643916       (a4)    */
        word 0xbf741839 0xeb88156e  /*   -0.0049059164525763043      (a3)    */
        word 0x3fa5be29 0x8adf0369  /*    0.042466448022884525       (a2)    */
        word 0xbfcf5e46 0x537ab906  /*   -0.24506453586713678        (a1)    */
        word 0x3fe6a09e 0x667f3bcc  /*    0.70710678118654746        (a0)    */
```

FIGURE A-1: Microcode assembler source for the evaluation of e^x

Once we obtain 2^{-t}, we just multiply it by 2^N to obtain e^x. Table A-1 shows the sequence of operations executed in the floating point unit ALU and multiplier, as well as the corresponding microcode address and the data flow in the XBus and YBus. Chapter 4 contains a detailed description of the Numerical Engine architecture and its floating point unit. Figure A-1 illustrates the microcode assembler source `exp.u`, used for the evaluation of e^x.

Assuming a FPU clock cycle of $20ns$, the computation of e^x takes 18 cycles or $360ns$. In this period, 32 floating point operations were executed, which corresponds to 89% utilization of the FPU, or 89 MFlops. For the sake of comparison, the IBM 3090 mainframe can compute e^x in $3.03\mu s$, while a 16 Mhz version of the 68881 floating point accelerator requires $31.3\mu s$. These results help support the hypothesis that a single Numerical Engine processor could attain a high degree of floating point utilization for the device evaluation tasks described in Chapter 3.

B

Emulation Tools

We will describe in this appendix the tool suite developed for the architectural emulation. The objective of these tools are providing documentation, system verification, and design trade-off analysis. As discussed before, the tool suite was built around `SIMLAB`, a circuit simulator similar to SPICE.

Figure B-1 shows the modified `SIMLAB` emulation environment. In addition to the original `SIMLAB` core, shown in dashed lines, several modules were incorporated to parse the architectural description of the system; perform behavioral and register-transfer emulations; schedule the multiprocessor operations; link, and load into the emulated microprogram memory, relocatable microcode modules.

A separate Microassembler program was developed for simplifying the generation of relocatable microcode object modules for the RTL emulation. The behavioral level emulation can incorporate new procedures by simply compiling and linking (with the host `cc` compiler) into the `SIMLAB` library.

Figure B-2 depicts a typical session of the modified `SIMLAB` program, in which the user configures the architectural database (`=> configuration hardware`), reads in the circuit netlist of a PLA circuit (`=> circuit decpla.rel`) to be simulated, and performs the simulation (`=> run`).

The `configuration` command invoked the architectural parser for reading in the user-selected file named `hardware`, containing a description of the architecture to be emulated. The parser first reads information contained in the architectural description file such as the number of processors, the size of the different memory subsystems in each processor, cost functions for the scheduling such as the latency of the floating point unit, and other environment variables. Then, the parser will set up a common database for both behavioral (BLS) and register-transfer (RTL) levels of emulation, configure the scheduler cost function database, and link and load in the microprogram memory of all processors that will be emulated.

Figure B-3 depicts a part of the architectural description file "`hardware`", loaded by the user in the beginning of the session shown in Figure B-2. The first set of bindings (`t_normalize`,

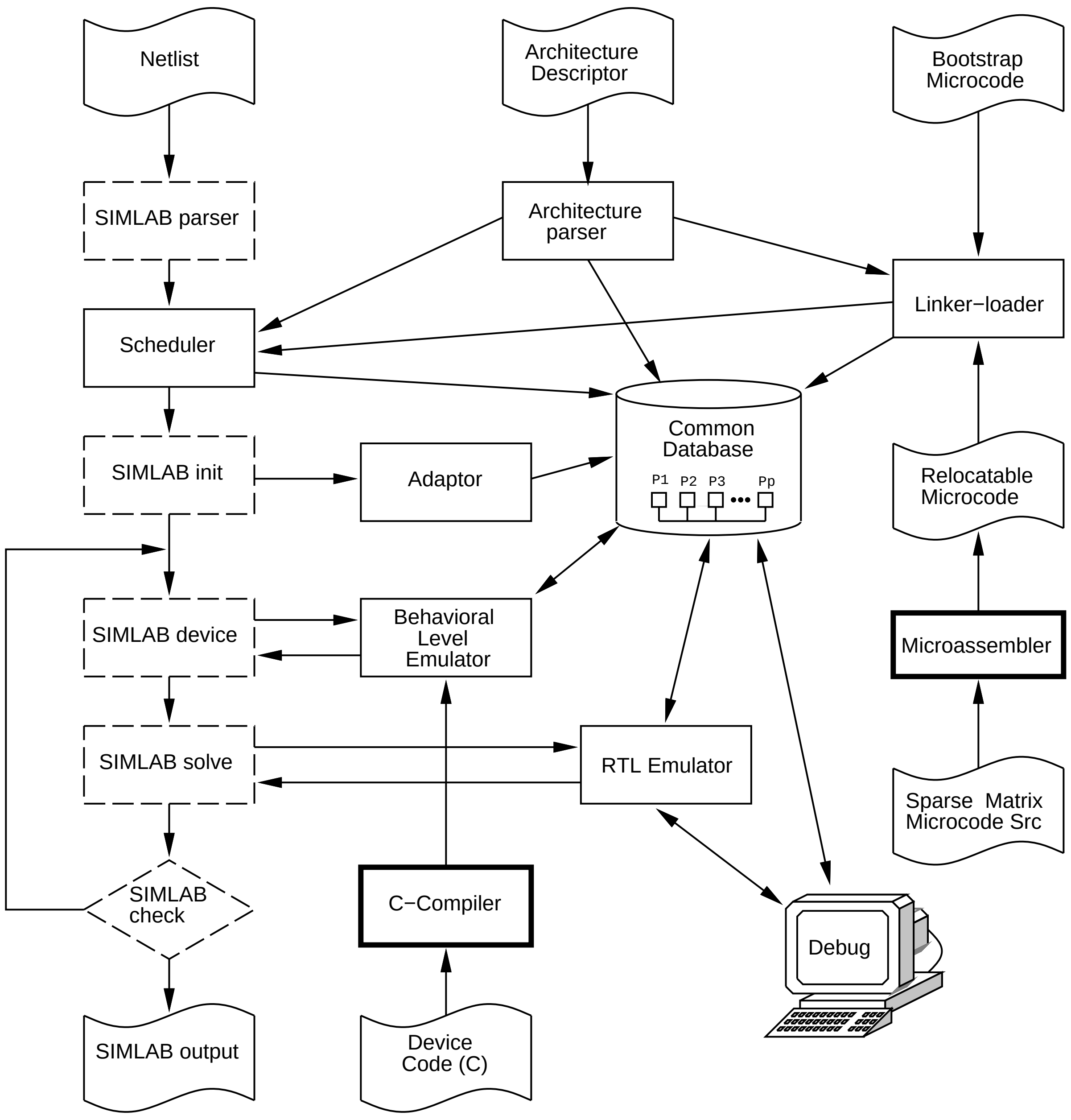


FIGURE B-1: The SIMLAB emulation environment

etc.) indicate special cost functions that are applied during the scheduling. For instance, when trying to determine the cost of the normalization of row i, the scheduler will first set the value of `size_source` to the number of elements after the diagonal of row i, and then call a system routine that uses the user-supplied `14 + 2 * size_source` cost function applied to the size of row i to compute the value of `t_normalize` to be returned to the scheduler. The second set of bindings are simpler, they just specify the numerical values of system-dependent variables like the number of processors (`processors`), the cost for evaluating a shared current source

```
                              < SIMUGLY 1.0 >
                 (c) 1990 Massachusetts Institute of Technology
                       Research Laboratory of Electronics
                       All Rights Reserved      15 Sep 93
=>
=> configuration hardware   % parses ``hardware'' for setting up emulation

Loading Microprogram Modules:
======= =========== ========
/home/gn/guest/ricardo/demos/microprogram/main.m
/home/gn/guest/ricardo/demos/microprogram/libsparse.a
......

=> circuit decpla.rel       % reads in the circuit netlist

Reading decpla.rel
stop = 2.500000e-07
cmin = 1.000000e-12
dodc = 0

=> run                      % starts circuit simulation

Simulating circuit decpla.rel
14 models, evaluation requires 164 memory words (1312 bytes)
LU + FB 27 levels, 323 tasks which cost 2043, maximum is 121
LU + FB needs 728 mults, 549 adds and 56 divs
Critical path costs 280
 1 processors, time=     2043, speedup=     1 ut=    1 (real=    1)
Processor  1 device evaluation cost: 32263

     Transient Simulation Results
       Wed May 26 13:48:23 1993
-----------------------------------------
                       Host: grits
          Simulation Method: Pointwise Solution
    Transient Solution Time: 46.7
         Integration Method: Trapezoidal
                  Timesteps: 100
  Nonlinear Solution Method: Newton
          Newton Iterations: 306
     Linear Solution Method: Direct
 Initial Decomposition Time: 0
       Linear Solution Time: 34.3167
Total RTL simulated  LUDec:  14
Total BLS simulated  LUDec:  118
Total R+B simulated  LUDec:  132
Total RTL simulated  FEBS:   31
Total BLS simulated  FEBS:   275
Total R+B simulated  FEBS:   306
LU Decomp:     38880 cycles, 0.0007776 (each: 2777 cycles, 5.55429e-05)
FE-BS:         29822 cycles, 0.00059644 (each: 962 cycles, 1.924e-05)
Solve:         68702 cycles, 0.00137404

=> quit                     % exits the simulation
```

FIGURE B-2: A Typical SIMLAB session

(evs_isrc) or a local piecewise-linear voltage source (evl_pwlsrc) as described in Section 3.2.1, the size of the index memory and its static column size (index_memory_size and page_size) as described in Chapter 4, or parameters like the κ (elasticity) as described in Section 2.3 and the size of the O^2SA structure, specified in multiples of the n, as described in Section 2.5. Finally, the architectural description file executes rtl_init to create the common database shown in Figure B-1 and to link-load the three user-selected relocatable microcode object files into the microprogram memory of the processors that will be emulated.

When the user starts the simulation, the scheduler will use the parameters and cost functions set by the user to generate a sequence of instructions for device evaluation and sparse matrix decomposition on the target multiprocessor. It also uses the global address entries specified in the relocatable microcode during the link-load phase to bind the microcode procedure call addresses with the code being generated. The log messages specifying the number of device models, the size of the DAG, the number of tasks, and the expected speedup printed in the beginning of the simulation are generated by the scheduler. Control is then passed to the main SIMLAB core, which will execute the circuit simulation. During the circuit simulation, the SIMLAB core will call the behavioral level emulator for assembling a system of network equations by computing the device contributions, and the RTL emulator for solving it using the microcode sparse matrix package. In order to save time, the parameter hardware_usage was set to ten in this particular simulation to substitute the rather costly RTL emulation of the solver by a faster behavioral emulation nine out of 10 times it is invoked. During the development of the sparse matrix solver, a simple microcode debugger with breakpoint and trace capabilities was extensively used inside the main loop of the circuit simulation to fix the sparse matrix microcode.

At the end of the simulation, SIMLAB will print the resulting waveforms to a file and will print a summary of the transient simulation results, with some statistics of RTL and BLS usage.

```
#include "../../home"

/* special scheduler, size-source dependent bindings */

t_normalize   <- 14 + 2 * size_source;
t_update      <- 2 + size_source;
t_forwelim    <- 1 + size_source;
t_backsub     <- 1 + size_source;
t_scatter     <- size_source;
t_busno       <- size_source;

/* normal, fixed  bindings for cost functions and settting variables */

processors=              1;
o2sa_density=            0.8;
o2sa_size=               3;
elast=                   10.0;
t_busfe=                 1;
t_busbs=                 1;
p_normalize=             12;
p_febs=                  1;
p_bus=                   1;
evl_isrc=                4;
evs_isrc=                2;
evl_pwlsrc=              30;
main_diode=              80;
stuff_diode=             8;
trans_diode=             6;
main_mos1=               100;
stuff_mos1=              10;
trans_mos1=              12;
bus_fifo_size=           512.0;
almost_full_size=        250.0;
destination_memory_size= 131072.0;
fifoz_size=              512.0;
index_memory_size=       524288.0;
memory_page_size=        2048.0;
rfile_memory_size=       64.0;
source_memory_size=      524288.0;
useq_memory_size=        4096.0;
hardware_usage=          10;

u1= HOME + "microprogram/main.m";
u2= HOME + "microprogram/libsparse.a";
u3= HOME + "microprogram/libsys.a";
rtl_init(u1,u2,u3,3);
```

Figure B-3: Architectural description file used in the SIMLAB session

Bibliography

[Agrawal92] P. Agrawal, J. Trotter, and R. Telichevesky, "PACE: A Multiprocessor System for VLSI Circuit Simulation," *AT&T Conference on Electronic Testing,* ACET 92, pp. 1031-1036, Princeton, October 1992.

[Bit90] *BIT 2130/3130/4130 Floating Point Unit*, Advance Information, Bipolar Integrated Technologies, Inc. 1990.

[Chang88] M. Chang and I.N. Hajj, "iPRIDE: A Parallel Integrated Circuit Simulator Using Direct Method," *IEEE International Conference on Computer Aided Design,* pp. 304-307, Nov. 1988.

[Chappell91] T.I. Chappell, B.A. Chappell, S.E. Schuster, J. W. Allan, S. P. Klepner, R. V. Joshi, R. L. Franch, "A 2-ns Cycle, 3.8-ns Access 512-kb CMOS ECL SRAM with a Fully Pipelined Architecture,", *IEEE J. of Solid-State Circuits,* Vol 26, No. 11, pp. 1577-1585, November 1991.

[Clenshaw62] C. W. Clenshaw, *Chebyshev series for mathematical functions*, Mathematical Tables Vol 5., National Physical Laboratory, H.M. Stationery Office, London, 1962.

[Clenshaw63] C. W. Clenshaw, G. F. Miller and M. Woodger, "Algorithms for Special Functions I", *Numerische Mathematik* 4, pp. 403-419, 1963.

[Dhillon91] I. S. Dhillon, N. K. Karmakar, K. G. Ramakrishnan, "An Overview of the Compilation Process for a New Parallel Architecture," *Proceedings of the Fifth Canadian Supercomputing Conference,*, Fredericton, N.B., Canada, June 1991.

[Dobberpuhl92] D. W. Dobberpuhl, R.T. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R. A. Conrad, D.E. Dever, B. Gieseke, S.M.N. Hassoun, G.W. Hoeppner, K. Kuchler, M. Ladd, B.M. Leary, L. Madden, E.J. McLellan, D.R. Meyer, J. Montanaro, D.A. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam, "A 200-MHz 64-b Dual-Issue CMOS Microprocessor," *IEEE*

Journal of Solid-State Circuits, Vol. 27, No. 11, pp. 1555-1567, November 1992.

[Dosaka92] K. Dosaka, Y. Konoshi, K. Hayano, K. Himukashi, A. Yamazaki, H. Iwamoto, M. Kumanoya, H. Hamano, and T. Yoshihara, "A 100-Mhz 4-Mb Cache DRAM with Fast Copy-Back Scheme," *IEEE Journal of Solid State Circuits,* pp. 1534-1539, November 1992.

[Duff86] I. S. Duff, A. M. Erisman, J. K. Reid, *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, England, 1986.

[Efe82] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer,* 15(6), 1982 pp. 50-56.

[Eisenstat77] S. C. Eisenstat, M. C. Gursky, M. H. Schultz and A. H. Sherman, *Yale Sparse Matrix Package II: The Nonsymmetric Codes*, Yale University Computer Science Department Research Report 114, 1977.

[Ellis85] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, ACM Doctoral Dissertation Award 1985, MIT Press, Cambridge, Mass 1986.

[Gerasoulis90] A. Gerasoulis, S. Venugopal, T. Yang, "Clustering Task Graphs for Message Passing Architectures," *1990 ACM International Conference on Supercomputing*, June 11-15, Amsterdam, 1990.

[Ginosar85] R. Ginosar and N.G. Jacobson, "The Simulation Machine: A VLSI Architecture for Circuit Simulation," *IEEE International Conference on Computer Design,* pp. 590-594, 1985.

[Golub89] G. H. Golub and C. F. van Loan, *Matrix Computations*, 2nd Edition, The Johns Hopkins University Press, Baltimore, Maryland 1989.

[Graham66] R. L. Graham, "Bounds for Certain Multiprocessing Anomalies," *Bell System Tech. J.* 45, pp. 1563-1581, 1966.

[Graham79] R. L. Graham, E. L. Lawler, J. K. Lenstra, A.H.G. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Annals of Discrete Mathematics,* 5, pp. 287-326, North Holland Publishing Company, 1979.

[Greengard87] L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, PhD. thesis, Yale University, April 1987. MIT Press, Cambridge, Mass. 1988.

[Gyurcsik85] R. S. Gyurcsik, D. O. Pederson, "A MOS Transistor Model-Evaluation Attached Processor for Circuit Simualation," *IEEE International Conference on Computer Aided Design,* pp. 234-236, 1985.

[Hu61] T. C. Hu, "Parallel sequencing and assembly line problems", *Operations Res.*, Vol 9, pp. 841-848, 1961.

[IDT92] Integrated Device Technology, Inc. "High Performance Logic Data Book," *Integrated Device Technology, Inc.*, 1992.

[Intel90] Intel "*Intel i860* 64-bit Microprocessor (Data Sheet)," *Intel,* 240296-002, April 1990.

[Jackson55] J. R. Jackson, "Scheduling a Production Line to Minimize Maximum Tardiness," *Research Report 43,* Management Science Research Project, University of California, Los Angeles, 1955.

[Huang79] J. W. Huang, O. Wing, "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Trans. on Circuits and Systems,* pp. 726-732, Sept. 1979.

[Kelley61] J. E. Kelley, "Critical-Path Planning and Scheduling: Mathematical Basis," *Operations Res.* **9**, No. 3, May 1961

[Kernighan70] B. W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical Journal,* pp. 291-307, Feb. 1970.

[Kim88] S. J. Kim, J. C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *International Conference on Parallel Processing,* 3, pp. 1-8, 1988.

[Kundert86] K. S. Kundert, "Sparse Matrix Techniques," *Circuit Analysis, Simulation and Design,* A.E. Ruehli (Editor), Elsevier Science Publishers B.V., North-Holland, pp. 281-324, 1986.

[Kundert88] K. S. Kundert, A. Sangiovanni-Vincentelli, *Sparse User's Guide*, Department of Electrical Engineering and Computer Sciences University of California, Berkeley, CA, April 1988.

[Lewis86] D. M. Lewis, "A High Performance Hardware Accelerator for Circuit Level Simulation of VLSI Circuits," *IEEE International Conference on Computer Aided Design,* pp. 386-389, 1986.

[Lumsdaine90] A. Lumsdaine, M. Silveira and J. White, *SIMLAB User's Guide*, Research Laboratory of Electronics, Massachusetts Institute of Technology, 1990.

[Malcolm59] D. G. Malcolm, J. H.Roseboom, C. E. Clark, W. Fazar, "Applications of a Technique for Research and Development Program Evaluation," *Operations Res.* **7**, No. 5, September 1959.

[Markowitz57] H. M. Markowitz, "The Elimination Form of the Inverse and its Application to Linear Programming", *Management Science*, Vol 3, pp. 255-269, April, 1957.

[Nagel75] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits", *Memorandum no. ERL-M520,* Electronics Research Laboratory, University of California, Berkeley, May 1975.

[Nakata87] T. Nakata, N. Tanobe, H. Onozuka, T. Kurobe, N. Koike, "A Multiprocessor System for Modular Circuit Simulation," *IEEE International Conference on Computer Aided Design,* pp. 364-367, 1987.

[Press92] W. H. Press, W. T. Vetterling, S. A. Teukolsky, B. P. Flannery, *Numerical Recipes in C — The Art of Scientific Computing,* Cambridge University Press, 1992.

[Rao88] Rao S., Ackland B., London T. and Hatamiam M., "Perfect Hashing for Sparse Matrix Elimination," Private Communication, July 1988.

[Rose78] D. J. Rose and R. E. Tarjan, "Algorithmic Aspects of Vertex Elimination on Directed Graphs", *SIAM J. Appl. Math.* **34**, pp. 176-197.

[Sadayappan88] P. Sadayappan and V. Visvanathan, "Parallelization and Performance Evaluation of Circuit Simulation on a Shared Memory Multiprocessor", *IEEE Trans. Comp,* Vol. 37, No. 12, Dec. 1988

[Sadayappan89] P. Sadayappan and V. Visvanathan, "Efficient Sparse Matrix Factorization for Circuit Simulation on Vector Supercomputers," *IEEE Trans. on Computer Aided Design,* pp. 1276-285, Vol. 8, No. 12, Dec. 1989

[Sarkar89] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multicomputers,* MIT Press, Cambridge, Mass, 1989.

[Smart89] D. Smart and J. White, "Reducing the Parallel Solution Time of Sparse Matrices using Reordered Gaussian Elimination and Relaxation,", *Private Communication*

[Stager87] L. K. Stager, "Vectorization of the LU-Decomposition for Circuit Simulation", *Proc. VLSI 87*, pp. 353-362, Vancouver, Canada, August 1987.

[Tarjan79] Tarjan, R. and Yao, A. "Storing a Sparse Table", *Comm ACM*, pp. 606-611, No. 11, Vol 22, November 1979.

[Telichevesky91a] R. Telichevesky, P. Agrawal, J. A. Trotter, "Partitioning Schemes for Circuit Simulation on a Multiprocessor Array", *International Conference on Application Specific Array Processors*, pp. 177-182, Barcelona, Spain, September 1991.

[Telichevesky91b] R. Telichevesky, P. Agrawal, J. A. Trotter, "A New $O(nlogn)$ Scheduling Heuristic for Parallel Decomposition of Sparse Matrices", *Internation Conference on Computer Design 91*, pp. 612-616, Cambridge, MA, October 1991.

[Touzeau84] R. F. Touzeau, "A Fortran Compiler for the FPS-164 scientific computer", *Proceedings of SIGPLAN 84 Symposium on Compiler Construction,*, pp. 48-57, ACM, June 1984.

[Trotter90a] J. A. Trotter and P. Agrawal, "Matrix Factorization Algorithms for a Distributed Memory Multiprocessor Architecture," *Private Communication,* May 1990.

[Trotter90b] J. A. Trotter and P. Agrawal, "Fast Overlapped Scattered Array Storage Schemes for Sparse Matrices," *IEEE International Conference on Computer Aided Design,* pp. 450-453, Nov. 1990.

[Vlach83] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design,* Van Norstrand Reinhold Company, New York, 1983.

[Wing80] O. Wing, J. W. Huang, "A Computation Model of Parallel Solution of Linear Equations", *IEEE Trans. on Computers,* pp. 632-638, July 1980.

[Yannakakis81] M. Yannakakis, "Computing the minimum fill-in is NP-complete", *SIAM J. Alg. Disc. Meth.* **2**, pp. 77-79

[Ziegler77] Ziegler S., "Smaller Faster Table Driven Parser", Madison Academic Computer Center, University of Wisconsin, Madison, WI, 1977. Unpublished work.